Dynamical Network Analysis

Marcelo C. R. Melo ; Rafael C. Bernardi

Aug 23, 2023

CONTENTS:

This package was built to provide an updated and enhanced Python implementation of the Dynamical Network Analysis method, for the analysis of Molecular Dynamics simulations. The package was optimized for interactive data analysis and visualization through Jupyter Notebooks (see *Tutorial*), and provides an interface for rendering publication-quality figures using VMD. It allows for extensive customization of analysis workflows to suit research-specific needs.

CHAPTER

ONE

INTRODUCTION

1.1 Scientific Background

In the last few decades molecular dynamics (MD) simulations have become an indispensable tool for mechanistic analysis in structural biology. From its first applications, revealing the fluid-like interior of protein that result from the diffusional character of local atomic motion, to more recent applications simulating entire organelles, the information content generated by MD studies has grown rapidly. With the increase of system sizes and the frequent use of enhanced sampling techniques, came the need for new and enhanced analysis tools, capable of extracting information from massive amounts of data and generating new insight.

Developed just over a decade ago, a particularly interesting technique that has recently become popular is the analysis of dynamical networks. This technique has been employed to study how groups of atoms interconnect in *communities*, and also the allosteric signaling in tRNA:protein complexes, glutamine amidotransferase, and many other systems.

The analysis of networks and their properties has a long history, with applications in diverse fields such as engineering, and social networks, and their approach to modelling molecular systems is particularly fruitful, leading to a rich field of research. Using MD simulations to extract dynamical features from biomolecules, from simple proteins to complexes, one can convert the atomic representation of the system into a *nodes-and-edges* representation that can then be analyzed much like any other graph. A key source of information is the partitioning of the network in subgroups (or communities) using algorithms such as Girvan-Newman's, providing information on cooperative motion within a protein's subdomains, or on residues that mediate communication between communities. Both are computationally challenging tasks, and can become very expensive as the size of the network grows.

1.2 Dynamical Network Analysis

The approach take in Dynamical Network Analysis is one based on the correlation of movement of representative atoms (or *nodes*), such as alpha-carbons of amino acid residues. This serves as a measurement to determine the existence and strength of a link between different atoms or molecules of a system.

Each node represents a set of atoms of a given residue, and there may be more than one node per residue. By default, amino acid residues are represented by a single node located in their alpha-carbons, and nucleotides by two nodes, one in the backbone phosphate, and one in the nitrogenous base. Water molecules have one node in their oxygen atom, and ions are trivially represented by one node.

The analysis performed here focuses solely on the correlation of movement of nodes in close proximity. This way, only short-range *direct* interactions are explicitly calculated, and long range interactions are determined through the analysis of the network itself.

To determine which nodes are in contact, the shortest distance between heavy atoms (all atoms excluding hydrogen atoms) represented by two nodes is calculated. If the distance is shorter than 4.5 Angstroms in a simulation frame, the pair of nodes is said to be in contact in that frame. If a pair of nodes is in contact in more than 75% of a simulation, they are considered to be in contact for the purposes of network analysis.

After determining all nodes in contact throughout a simulation, and calculating the correlation of motion between them, the network can then be visualized (see *Image 1*).

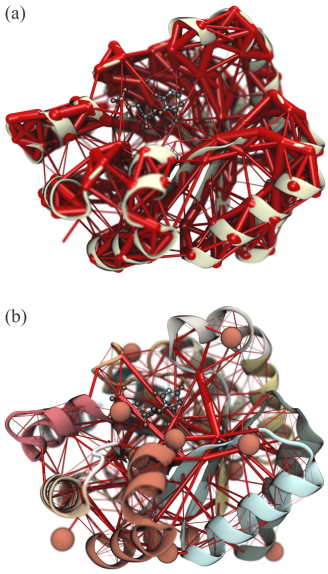

Fig. 1: Networks analysis of OMP-decarboxylase. (a) Full network revealing the most correlated regions of the enzyme. The weight of the network edges (represented by thickness of red tubes) is given by its normalized generalized correlation coefficient. (b) Rendering showing communities and betweenness values of edges of the OMP-decarboxylase dynamic network. Communities are delineated by the different colors of the protein secondary structure, while betweenness values of network edges are indicated by the thickness red tubes. Both images were rendered with VMD using the new Network Viewer 2.0 GUI.

1.3 Current Implementation

This package was built to provide all functionalities necessary to the analysis of MD simulations using the Dynamical Network Analysis method. In particular, it accelerates the calcualtion of generalized correlation coefficients by first calculating a contact matrix for the network, and then parallelizing the correlation calculation only for the pairs of nodes in contact. This can save over 99% of the computational cost of generalized correlation calculations.

CHAPTER

TWO

INSTALLATION

2.1 Installing with *pip*

To install this package and all required Python packages, simply run:

```
$ pip install dynetan
```

2.2 Requirements

The core package **requires** Python 3.9 or greater and the following python dependencies:

- MDAnalysis
- SciPy
- NumPy
- pandas
- networkx
- numba
- h5py
- python-louvain

The following packages are not necessary for the core functionality, but are suggested for use along with jupyter notebooks:

- ipywidgets
- colorama
- nglview
- rpy2
- tzlocal

2.3 Build the package from source:

Ensure pip, setuptools, and wheel are up-to-date:

```
$ python -m pip install --upgrade pip build
```

Create a Wheel file locally:

```
$ python3 -m setup.py build
```

2.4 Troubleshooting installation with *pip*

If during the installation process with *pip* you find the error *fatal error: Python.h: No such file or directory*, make sure your Linux distribution has the necessary development packages for Python. They contain header files that are needed for the compilation of packages such as MDAnalysis. These packages will be listed as "python3-dev" or similar. For example, in Fedora 32, one can use the command *dnf install python3-devel* to install additional system packages with Python headers.

Similarly, if during the installation process you find the error *gcc: fatal error: cannot execute 'cc1plus': execvp: No such file or directory*, make sure you have development tools for gcc and c++. For example, in Fedora 32, one can use the command *dnf install gcc-c++* to install additional system packages with c++ development tools.

CHAPTER

THREE

TUTORIAL

A series of tutorials were designed to provide a detailed description of the workflow of the Dynamical Network Analysis implementation provided here. The tutorials combine system preparation, data generation, and analysis tools provided in this package. Tutorials also cover the use if this package through interactive interface using Jupyter Notebooks, as well as command-line-interface (CLI) scripts that can be deployed for execution in remote computer clusters.

For the latest version of the tutorial, download the tutorial files here along with accompanying trajectory data here (trajectory files are approximately 500MB in size).

CHAPTER

FOUR

USAGE

This package was built to provide an updated and enhanced Python implementation of the Dynamical Network Analysis method, for the analysis of Molecular Dynamics simulations. The package was optimized for both interactive use through Jupyter Notebooks (see *Tutorial*) and for command-line-interface use (such as in scripts for remote execution. The package allows for extensive customization of analysis to suit research-specific needs.

We present below an overview of the process of analysing an MD simulation, using the OMP decarboxylase example that was examined in the reference publication (see *Citing*) and the associated *Tutorial*.

Load your simulation data by creating a *DNAproc* object:

```
# Load the python package
import os
import dynetan

# Create the object that processes MD trajectories.
dnap = DNAproc()
```

Select the location of simulation files:

```
# Path where input files will searched and results be written.
workDir = "./TutorialData/"

# PSF file name
psfFile = os.path.join(workDir, "decarboxylase.0.psf")

# DCD file name
dcdFiles = [os.path.join(workDir, "decarboxylase.1.dcd")]
```

Select the number of windows into which your trajectory will be split. This can correspond to a long contiguous simulation or multiple independent concatenated replicas of the same system:

```
# Number of windows created from full simulation.
numWinds = 4

# Sampled frames per window (for detection of structural waters)
numSampledFrames = 10
```

Select a ligand to be analysed and segment IDs for the biomolecules to be studied. For automatic detection of structural water molecules, provide the name of the solvent residue:

```
ligandSegID = "OMP"
```

(continues on next page)

(continued from previous page)

```
# Segment IDs for regions that will be studied.
segIDs = ["OMP","ENZY"]

# Residue name for solvent molecule(s)
h2oName = ["TIP3"]
```

Set the node groups for user-defined residues:

```
# Network Analysis will make one node per protein residue (in the alpha carbon)
# For other residues, the user must specify atom(s) that will represent a node.
customResNodes = {}
customResNodes["TIP3"] = ["OH2"]
customResNodes["OMP"] = ["N1","P"]

# We also need to know the heavy atoms that compose each node group.

usrNodeGroups = {}

usrNodeGroups["TIP3"] = {}
usrNodeGroups["TIP3"]["OH2"] = set("OH2 H1 H2".split())

usrNodeGroups["OMP"] = {}
usrNodeGroups["OMP"]["N1"] = set("N1 C2 O2 N3 C4 O4 C5 C6 C7 OA OB".split())
usrNodeGroups["OMP"]["P"] = set("P OP1 OP2 OP3 O5' C5' C4' O4' C1' C3' C2' O2' O3'".
↪split())
```

Define the parameters that will control contact detection:

```
# Cutoff for contact map (In Angstroms)
cutoffDist = 4.5

# Minimum contact persistance (In ratio of total trajectory frames)
contactPersistence = 0.75
```

Finally, load all data to the *DNAproc* object:

```
### Load info to object

dnap.setNumWinds(numWinds)
dnap.setNumSampledFrames(numSampledFrames)
dnap.setCutoffDist(cutoffDist)
dnap.setContactPersistence(contactPersistence)
dnap.seth2oName(h2oName)
dnap.setSegIDs(segIDs)

dnap.setCustomResNodes(customResNodes)
dnap.setUsrNodeGroups(usrNodeGroups)
```

In its simplest form, the code will load the MD simulation, detect structural water molecules, and create a network representation of the nodes selected so far:

```
dnap.loadSystem(psfFile,dcdFiles)
```

(continues on next page)

(continued from previous page)

```
dnap.selectSystem(withSolvent=True)

dnap.prepareNetwork()
```

After the nodes and node groups are selected, the system is aligned, contacts are detected, and the calculation of correlation coefficients can begin:

```
dnap.alignTraj(inMemory=True)

dnap.findContacts(stride=1)

dnap.calcCor(ncores=1)
```

With the correlation matrix of each simulation window, we create graph representations for each simulation window, and calculate network properties such as optimal paths, betweenness and communities:

```
dnap.calcGraphInfo()

dnap.calcOptPaths(ncores=1)

dnap.calcBetween(ncores=1)

dnap.calcCommunities()
```

To automate the detection of edges between two separate subunits of a biomolecular complex, we can specify segment IDs and request the identification of interface connections:

```
dnap.interfaceAnalysis(selAstr="segid ENZY", selBstr="segid OMP")
```

Finally, all data can be saved to disk:

```
dnap.saveData(fullPathRoot)
```

All the interactive visualization of the structure and network nodes and edges, optimal paths, communities, and high resolution rendering are performed through jupyter notebooks. Please refer to the *Tutorial* for detailed examples.

CHAPTER

FIVE

CITING

To cite this package please use the following publication:

- Generalized correlation-based dynamical network analysis: a new high-performance approach for identifying allosteric communications in molecular dynamics trajectories. JCP (2020). DOI: 10.1063/5.0018980

For further discussion and scientific background, please refer to:

- Experimental and computational determination of tRNA dynamics. FEBS Letters (2010). DOI: 10.1016/j.febslet.2009.11.061
- Exit strategies for charged tRNA from GluRS. JMB (2010). DOI: 10.1016/j.jmb.2010.02.003
- Dynamical Networks in tRNA:protein complexes. PNAS (2009). DOI: 10.1073/pnas.0810961106

CHAPTER

SIX

REFERENCE

The documentation provided here (attempts to) follow the Google style of code documentation, and is built using Sphinx and its Napoleon module.

6.1 Process Trajectory Data

This dedicated class controls all trajectory processing necessary for Dynamical Network Analysis.

class dynetan.proctraj.**DNAproc**(*notebookMode=True*)

The Dynamic Network Analysis processing class contains the infrastructure to carry out the data analysis in a DNA study of a biomolecular system.

This class uses optimized auxiliary functions to parallelize the most time-consuming aspects of Dynamical Network Analysis, including contact detection, calculation of correlation coefficients, and network properties.

The infrastructure built here is also focused on combining multiple molecular dynamics simulations of the same system, emphasizing the calculation of statistical properties. This allows the comparison between replicas of the same biomolecular system or time evolution of a particular system.

alignTraj(*selectStr=''*, *inMemory=True*, *verbose=0*)

Wrapper function for MDAnalysis trajectory alignment tool.

Parameters

- **inMemory** (*bool*) – Controls if MDAnalysis *AlignTraj* will run in memory.
- **selectStr** (*str*) – User defined selection for alignment. If empty, will use default: Select all user-defined segments and exclude hydrogen atoms.
- **verbose** (*int*) – Controls verbosity level.

Returns
none

Return type
None

calcBetween(*ncores=1*)

Main interface for betweeness calculations.

Calculates betweenness for all nodes in the network using NetworkX implementation of the betweenness centrality for edges and eigenvector centrality for nodes. When using more than one core, this function uses Python's *multiprocessing* infrastructure to calculate betweenness in multiple simulation windows simultaneously.

Note: See also *calcBetweenPar()*.

Parameters
ncores (*int*) – Defines how many cores will be used for calculation. Set to *1* in order to use the serial implementation.

Return type
None

calcCartesian(*backend='serial'*, *verbose=1*, *n_cores=1*)

Main interface for calculation of cartesian distances.

Determines the shortest cartesian distance between atoms in node groups of all network nodes. Using a sampling of simulation frames, the function also calculates statistics on such measures, including mean distance, standard error of the mean, minimum, and maximum. This allows analysis comparing network distances and cartesian distances.

Note: See also *calc_distances()* and *getCartDist()*.

Parameters

- **backend** (*str*) – Defines which MDAnalysis backend will be used for calculation of cartesian distances. Options are *serial* or *openmp*. This option is ignored if the distance mode is not "all".
- **verbose** (*int*) – Defines verbosity of output.
- **n_cores** (*int*) – Number of cores used to process cartesian distance between network nodes.

Return type
None

calcCommunities()

Calculate node communities using Louvain heuristics.

The function produces sets of nodes that are strongly connected, presenting high correlation coefficients.

It uses Louvain heuristics as an efficient and precise alternative to the classical Girvan–Newman algorithm, which requires much more computing power for large and highly connected networks. This method also maximizes the modularity of the network. It is inherently random, so different calculations performed on the same network data may produce slightly different results.

For more details, see the original reference.

Return type
None

calcCor(*ncores=1*, *forceCalc=False*, *verbose=0*)

Main interface for correlation calculation.

Calculates generalized correlation coefficients either in serial or in parallel implementations using Python's multiprocessing package. This function wraps the creation of temporary variables in allocates the necessary NumPy arrays for accelerated performance of MDAnalysis algorithms.

Note: See also *prep_mi_c()*.

Note: See also *calc_mir_numba_2var()*.

Note: See also *calc_cor_proc()*.

Parameters

- **ncores** (*int*) – Defines how many cores will be used for calculation of generalized correlation coefficients. Set to *1* in order to use the serial implementation.
- **forceCalc** (*bool*) – Defines if correlations will be calculated again even if they have been calculated before.
- **verbose** (*int*) –

Return type
None

calcEigenCentral()

Wrapper for calculation of node centrality.

Calculates node centrality for all nodes in all simulation windows. This calculation is relatively inexpensive and is only implemented for serial processing.

All results are stored in the network graph itself.

Return type
None

calcGraphInfo()

Create a graph from the correlation matrix.

Uses NetworkX to create a graph representation of the network. One graph is created per simulation window.

For network analysis, node *distances* are generated with a log transformation of the correlation values. This way, edges between nodes with higher correlation coefficients are considered "closer", with shorter distances, and nodes with low correlation coefficients are "far apart", with larger distance.

Note: See also *calcOptPathPar()* and *calcBetweenPar()*.

Return type
None

calcOptPaths(*ncores=1*)

Main interface for optimal path calculations.

Calculates optimal paths between all nodes in the network using NetworkX implementation of the Floyd Warshall algorithm. When using more than one core, this function uses Python's *multiprocessing* infrastructure to calculate optimal paths in multiple simulation windows simultaneously.

Note: See also *calcOptPathPar()*.

Parameters
ncores (*int*) – Defines how many cores will be used for calculation of optimal paths. Set to *1* in order to use the serial implementation.

Return type
None

checkContactMat(*verbose=1*)

Sanity checks for contact matrix for all windows.

Checks if the contact matrix is symmetric and if there are any nodes that make no contacts to any other nodes across all windows. The function also calculates the percentage of nodes in contact over the entire system.

Parameters
verbose (*bool*) – Controls how much output will the function print.

Return type
None

checkSystem()

Performs a series of sanity checks.

This function checks if the user-defined data and loaded simulation data are complete and compatible. This will print a series of diagnostic messages that should be used to verify if all calculations are set up as desired.

Return type
None

filterContacts(*notSameRes=True*, *notConsecutiveRes=False*, *removeIsolatedNodes=True*, *verbose=1*)

Filters network contacts over the system.

The function removes edges and nodes in preparation for network analysis. Traditionally, edges between nodes within the same residue are removed, as well as edges between nodes in consecutive residues within the same polymer chain (protein or nucleic acid). Essentially, nodes that have covalent bonds connecting their node groups can bias the analysis and hide important non-bonded interactions.

The function also removes nodes that are isolated and make no contacts with any other nodes. Examples are ions or solvent residues that were initially included in the system through the preliminary automated solvent detection routine, but did not reach the contact threshold for being part of the final system.

After filtering nodes and edges, the function updates the MDAnalysis universe and network data.

Parameters

- **notSameRes** (*bool*) – Remove contacts between nodes in the same residue.
- **notConsecutiveRes** (*bool*) – Remove contacts between nodes in consecutive residues.
- **removeIsolatedNodes** (*bool*) – Remove nodes with no contacts.
- **verbose** (*bool*) – Controls verbosity of output.

Return type
None

findContacts(*stride=1*, *verbose=1*, *n_cores=1*)

Finds all nodes in contact.

This is the main user interface access to calculate nodes in contact. This function automatically splits the whole trajectory into windows, allocates NumPy array objects to speed up calculations, and leverages MDAnalysis parallel implementation to determine atom distances.

After determining which frames of a trajectory window show contacts between atom groups, it checks the contact cutoff to determine if two nodes has enough contact during a simulation window to be considered "in contact". A final contact matrix is created and stored in the DNAproc object.

This function automatically updates the unified contact matrix that displays nodes in contact in *any* simulation window. The function also performs general sanity checks by calling *checkContactMat()*.

Parameters

- **stride** (*int*) – Controls how many trajectory frames will be skipped during contact calculation.
- **verbose** (*int*) – Controls verbosity level in the function.
- **n_cores** (*int*) – Number of cores used to process cartesian distance between network nodes.

Return type
None

getDegreeDict(*window=0*)

Compiles a dictionary with node degrees.

This wrapper function uses NetworkX graph object to list the degrees of all nodes.

Parameters
window (*int*) – Simulation window.

Return type
dict

getPath(*node_i*, *node_j*, *window=0*)

Wrapper for NetworkX reconstruct_path.

The function calls NetworkX's *reconstruct_path* to return the list of nodes that connect *nodeI* to *nodeJ*. This function must only be called **after** a path detection run has been completed (see *calcOptPaths()*).

Parameters

- **node_i** (*int*) – Node ID.
- **node_j** (*int*) – Node ID.
- **window** (*int*) – Simulation window.

Returns
List of node IDs.

Return type
list

getU()

Return MDAnalysis universe object.

Return type
Any

interfaceAnalysis(*selAstr*, *selBstr*, *betweenDist=15.0*, *samples=10*, *verbose=0*)

Detects interface between molecules.

Based on user-defined atom selections, the function detects residues (and their network nodes) that are close to the interface between both atom selections. That may include amino acids in the interface, as well as ligands, waters and ions.

Only nodes that have edges to nodes on the side of the interface are selected.

Using a sampling of simulation frames assures that transient contacts will be detected by this analysis.

Parameters

- **selAstr** (*str*) – Atom selection.
- **selBstr** (*str*) – Atom selection.
- **betweenDist** (*float*) – Cutoff distance for selection of atoms that are within *betweenDist* from both selections.
- **samples** (*int*) – Number of frames to be sampled for detection of interface residues.
- **verbose** (*int*) – Controls verbosity of output.

Returns
Number of unique nodes in interface node pairs.

Return type
int

loadSystem(*str_fn*, *traj_fns*)

Loads Structure and Trajectory files to an MDAnalysis universe.

Parameters

- **str_fn** (*str*) – Path to structure file, such as a PSF, PDB, Gro, or other file formats accepted by MDAnalysis.
- **traj_fns** (*str | List(str)*) – Path to one or more trajectory files. MDAnalysis will automatically concatenate trajectories if multiple files are passed.

Return type
None

prep_node_groups(*autocomp_groups=True*)

Prepare node groups and check system for unknown residues

This function will load the user-defined node groups into this object and will create node groups from standard proteic residues and trivial single-atom residues such as ions.

Parameters
autocomp_groups (*bool*) – Method will automatically add atoms from residues with defined node groups, as long as the atom is bound to another atom included in a node group. This is intended to facilitate the inclusion of hydrogen atoms to node groups without hard coded user definitions.

Returns
none

Return type
None

prepareNetwork(*verbose=0*, *autocomp_groups=True*)

Prepare network representation of the system.

Checks if we know how to treat all types of residues in the final system selection. Every residue will generate one or more nodes in the final network. This function also processes and stores the groups of atoms that define each node group in specialized data structures.

Note: We need this special treatment because the residue information in the topology file may list atoms in an order that separates atoms from the same node group. Even though atoms belonging to the same residue are contiguous, atoms in our arbitrary node groups need not be contiguous. Since amino acids have just one node, they will have just one range of atoms but nucleotides and other residues may be different.

Parameters

- **verbose** (*int*) –
- **autocomp_groups** (*bool*) –

Return type
None

saveData(*file_name_root='dnaData'*)

Save all network analysis data to file.

This function automates the creation of a *DNAdata* object, the placement of data in the object, and the call to its *saveToFile()* function.

Parameters
file_name_root (*str*) – Root of the multiple data files to be writen.

Return type
None

saveReducedTraj(*file_name_root='dnaData'*, *stride=1*)

Save a reduced trajectory to file.

This function automates the creation of a reduced DCD trajectory file keeping only the atoms used for Dynamical Network Analysis. It also creates a matching PDB file to maintain atom and residue names.

Parameters

- **file_name_root** (*str*) – Root of the trajectory and structure files to be writen.
- **stride** (*int*) – Stride used to write the trajectory file.

Return type
None

selectSystem(*withSolvent=False*, *inputSelStr=''*, *verbose=0*)

Selects all atoms used to define node groups.

Creates a final selection of atoms based on the user-defined residues and node groups. This function also automates solvent and ion detection, for residues that make significant contacts with network nodes. Examples are structural water molecules and ions.

This function will automatically remove all hydrogen atoms from the system, since they are not used to detect contacts or to calculate correlations. The standard selection string used is "not (name H* or name [123]H*)"

Ultimately, an MDAnalysis universe is created with the necessary simulation data, reducing the amount of memory used by subsequent analysis.

Parameters

- **withSolvent** (*bool*) – Controls if the function will try to automatically detect solvent molecules.
- **inputSelStr** (*str*) – Uses a user-defined selection for the system. This disables automatic detection of solvent/ions/lipids and other residues that may have transient contact with the target system.
- **verbose** (*int*) – Controls the verbosity of output.

Return type
None

setContactPersistence(*contact_persistence=0.75*)

Set contact persistence cutoff for contact detection.

Parameters
contact_persistence (*float*) – Ratio of total trajectory frames needed to consider a pair of nodes to be in contact. Usually set to 0.75 (75% of total trajectory).

Return type
None

setCustomResNodes(*customResNodes*)

Set atoms that will represent nodes in user defined residues.

Note: THIS METHOD HAS BEEN DEPRECATED. It has been fully replaced by *setNodeGroups()*.

Parameters
customResNodes (*dict*) – Dictionary mapping residue names with lists of atom names that will represent network nodes.

Return type
None

setCutoffDist(*cutoff_dist=4.5*)

Set cartesian distance cutoff for contact detection.

For all atom simulations, assuming only heavy atoms (non-hydrogen atoms) were kept in the system, this number is usually set to 4.5 Angstroms.

Parameters
cutoff_dist (*float*) – Cutoff distance for contact detection.

Return type
None

setDistanceMode(*mode='all'*)

Set the distance calculation method to find nodes in contact.

The supported options are:

1. `all`, which calculates all-to-all distances between selected atoms in the system.
2. `capped`, which uses a kdtree algorithm to only calculate distances between atoms closer than the network distance cutoff.

The "all" option will be faster for smaller systems. The "capped" option will benefit larger systems as it will require less memory.

Note: See also *setCutoffDist()*.

Parameters
mode (*str*) – Distance calculation mode.

Return type
None

setNodeGroups(*node_groups*)

Set atoms that will represent node groups in user-defined residues.

Network Analysis will create one network node per standard amino acid residue (in the alpha carbon). For other residues, the user must specify atom(s) that will represent a node. This function is used to define the heavy atoms that compose each node group for user-defined nodes.

Parameters
node_groups (*dict*) – Nested dictionary mapping residue names with atom names that will represent network nodes, and sets of heavy atoms used to define node groups.

Return type
None

setNumSampledFrames(*n_smpld_frms=1*)

Set number of frames to be sampled for solvent detection.

This will determine how many frames will be sampled for solvent detection per window, and for estimation of cartesian distance between node groups.

Parameters
n_smpld_frms (*int*) – Number of sampled frames per window.

Return type
None

setNumWinds(*num_winds=1*)

Set number of windows.

This will determine the number of windows into which the trajectory will be split.

Usage tip: If there are several concatenated replicas of the same system, make sure all have the same number of frames so that the split will extract each replica in a different window.

Parameters
num_winds (*int*) – Number of windows.

Return type
None

setSegIDs(*seg_ids*)

Set segment IDs for biomolecules ot be analyzed.

Parameters
seg_ids (*list*) – List of Segment IDs to be included in network analysis.

Return type
None

setSolvNames(*solvent_names*)

Set name of solvent molecule residue.

Parameters
solvent_names (*list*) – List of residue names used as solvent.

Return type
None

seth2oName(*solvent_names*)

Set name of solvent molecule residue.

Parameters
solvent_names (*list*) – List of residue names used as solvent.

Return type
None

6.2 Save and Load Data

This dedicated class stores and recovers results from Dynamical Network Analysis.

class dynetan.datastorage.**DNAdata**

Data storage and management class.

The Dynamic Network Analysis data class contains the infrastructure to save all data required to analyze and reproduce a DNA study of a biomolecular system.

The essential network and correlation coefficients data is stored in an HDF5 file using the H5Py module, allowing long term storage. The remaining data is stored in NumPy binary format and the NetworkX graph objects are stores in a pickle format. This is not intended for term storage, but the data can be easily recovered from the HDF5 data.

loadFromFile(*file_name_root*)

Function that loads all the data stored in a DNAdata object.

Parameters
file_name_root (*str*) – Root of the multiple data files to be loaded.

Return type
None

saveToFile(*file_name_root*)

Function that saves all the data stored in a DNAdata object.

Parameters
file_name_root (*str*) – Root of the multiple data files to be writen.

Return type
None

6.3 Contact Detection

This module contains auxiliary functions for the parallel calculation of node contacts.

dynetan.contact.**atm_to_node_dist**(*num_nodes*, *n_atoms*, *tmp_dists*, *atom_to_node*, *node_group_indices_np*, *node_group_indices_np_aux*, *node_dists*)

Translates MDAnalysis distance calculation to node distance matrix.

This function is JIT compiled by Numba to optimize the search for shortest cartesian distances between atoms in different node groups . It relies on the results of MDAnalysis' distance calculation, stored in a 1D NumPy array of shape (n*(n-1)/2,), which acts as an unwrapped triangular matrix.

The pre-allocated triangular matrix passed as an argument to this function is used to store the shortest cartesian distance between each pair of nodes.

This is intended as an analysis tool to allow the comparison of network distances and cartesian distances. It is similar to *dist_to_contact()*, which is optimized for contact detection.

Parameters

- **num_nodes** (*int*) – Number of nodes in the system.
- **n_atoms** (*int*) – Number of atoms in atom groups represented by system nodes. Usually hydrogen atoms are not included in contact detection, and are not present in atom groups.
- **tmp_dists** (*Any*) – Temporary pre-allocated NumPy array with atom distances. This is the result of MDAnalysis *self_distance_array* calculation.
- **atom_to_node** (*Any*) – NumPy array that maps atoms in atom groups to their respective nodes.
- **node_group_indices_np** (*Any*) – NumPy array with atom indices for all atoms in each node group.
- **node_group_indices_np_aux** (*Any*) – Auxiliary NumPy array with the indices of the first atom in each atom group, as listed in *node_group_indices_np*.
- **node_dists** (*Any*) – Pre-allocated array to store cartesian distances between *nodes*. This is a linearized upper triangular matrix.

Return type
None

dynetan.contact.**atm_to_node_dist_par**(*num_nodes*, *n_atoms*, *tmp_dists*, *atom_to_node*, *node_group_indices_np*, *node_group_indices_np_aux*, *node_dists*, *n_cores*)

(PARALLEL) Translates MDAnalysis distance calculation to node distance matrix.

This function is a parallel version of *atm_to_node_dist()*. It prepares the calculations and initialized *n_cores* processes using the *atm_to_node_dist_proc()* function.

Parameters

- **num_nodes** (*int*) – Number of nodes in the system.
- **n_atoms** (*int*) – Number of atoms in atom groups represented by system nodes. Usually hydrogen atoms are not included in contact detection, and are not present in atom groups.
- **tmp_dists** (*Any*) – Temporary pre-allocated NumPy array with atom distances. This is the result of MDAnalysis *self_distance_array* calculation.
- **atom_to_node** (*Any*) – NumPy array that maps atoms in atom groups to their respective nodes.

- **node_group_indices_np** (*Any*) – NumPy array with atom indices for all atoms in each node group.
- **node_group_indices_np_aux** (*Any*) – Auxiliary NumPy array with the indices of the first atom in each atom group, as listed in *node_group_indices_np*.
- **node_dists** (*Any*) – Pre-allocated array to store cartesian distances between *nodes*. This is a linearized upper triangular matrix.
- **n_cores** (*int*) – Number of cores used to process cartesian distance between network nodes.

Return type
None

dynetan.contact.**atm_to_node_dist_proc**(*num_nodes*, *n_atoms*, *tmp_dists*, *atom_to_node*, *node_group_indices_np*, *node_group_indices_np_aux*, *in_queue*, *out_queue*)

(PROCESS) Translates MDAnalysis distance calculation to node distance matrix.

This function is a parallel version of *atm_to_node_dist()*. It executes the calculations prepared by the *atm_to_node_dist_par()* function.

Parameters

- **num_nodes** (*int*) – Number of nodes in the system.
- **n_atoms** (*int*) – Number of atoms in atom groups represented by system nodes. Usually hydrogen atoms are not included in contact detection, and are not present in atom groups.
- **tmp_dists** (*Any*) – Temporary pre-allocated NumPy array with atom distances. This is the result of MDAnalysis *self_distance_array* calculation.
- **atom_to_node** (*Any*) – NumPy array that maps atoms in atom groups to their respective nodes.
- **node_group_indices_np** (*Any*) – NumPy array with atom indices for all atoms in each node group.
- **node_group_indices_np_aux** (*Any*) – Auxiliary NumPy array with the indices of the first atom in each atom group, as listed in *node_group_indices_np*.
- **in_queue** (*Any*) – Multiprocessing queue object for acquiring jobs.
- **out_queue** (*Any*) – Multiprocessing queue object for placing results.

Return type
None

dynetan.contact.**calc_distances**(*selection*, *num_nodes*, *n_atoms*, *atom_to_node*, *cutoff_dist*, *node_group_indices_np*, *node_group_indices_np_aux*, *node_dists*, *backend='serial'*, *dist_mode=0*, *verbose=0*, *n_cores=1*)

Executes MDAnalysis atom distance calculation and node cartesian distance calculation.

This function is a wrapper for two optimized atomic distance calculation and node distance calculation calls. The first is one of MDAnalysis' atom distance calculation functions (either *self_distance_array* or *self_capped_distance*). The second is the internal *atm_to_node_dist()*. All results are stored in pre-allocated NumPy arrays.

This is intended as an analysis tool to allow the comparison of network distances and cartesian distances. It is similar to *get_contacts_c()*, which is optimized for contact detection.

Parameters

- **selection** (*str*) – Atom selection for the system being analyzed.

- **num_nodes** (*int*) – Number of nodes in the system.
- **n_atoms** (*int*) – Number of atoms in atom groups represented by system nodes. Usually hydrogen atoms are not included in contact detection, and are not present in atom groups.
- **atom_to_node** (*Any*) – NumPy array that maps atoms in atom groups to their respective nodes.
- **cutoff_dist** (*float*) – Distance cutoff used to capp distance calculations.
- **node_group_indices_np** (*Any*) – NumPy array with atom indices for all atoms in each node group.
- **node_group_indices_np_aux** (*Any*) – Auxiliary NumPy array with the indices of the first atom in each atom group, as listed in *nodeGroupIndicesNP*.
- **node_dists** (*Any*) – Pre-allocated array to store cartesian distances.
- **backend** (*str*) – Controls how MDAnalysis will perform its distance calculations. Options are *serial* and *openmp*. This option is ignored if the distance mode is not "all".
- **dist_mode** (*int*) – Distance calculation method. Options are 0 (for mode "all") and 1 (for mode "capped").
- **verbose** (*int*) – Controls informational output.
- **n_cores** (*int*) – Number of cores used to process cartesian distance between network nodes.

Return type
None

dynetan.contact.**dist_to_contact**(*num_nodes*, *n_atoms*, *cutoff_dist*, *tmp_dists*, *tmp_dists_atms*, *contact_mat*, *atom_to_node*, *node_group_indices_np*, *node_group_indices_np_aux*)

Translates MDAnalysis distance calculation to node contact matrix.

This function is JIT compiled with Numba to optimize the search for nodes in contact. It relies on the results of MDAnalysis' distance calculation, stored in a 1D NumPy array of shape (n*(n-1)/2,), which acts as an unwrapped triangular matrix.

In this function, the distances between all atoms in an atom groups of all pairs of nodes are verified to check if any pair of atoms were closer than a cutoff distance. This is done for all pairs of nodes in the system, and all frames in the trajectory. The pre-allocated contact matrix passed as an argument to this function is used to store the number of frames where each pair of nodes had at least one contact.

This function DOES NOT store the shortest cartesian distances between node groups, it only checks if at least one pair of atoms from each group is close enough to count as a contact in a frame.

The *calc_distances()* function is a variation of this function. It calculates and stores the shortest cartesian distance between atoms of two groups, and does that for all node groups in the system.

Parameters

- **num_nodes** (*int*) – Number of nodes in the system.
- **n_atoms** (*int*) – Number of atoms in atom groups represented by system nodes. Usually hydrogen atoms are not included in contact detection, and are not present in atom groups.
- **cutoff_dist** (*float*) – Distance at which atoms are no longer considered 'in contact'.
- **tmp_dists** (*Any*) – Temporary pre-allocated NumPy array with atom distances. This is the result of MDAnalysis *self_distance_array* calculation.
- **tmp_dists_atms** (*Any*) – Temporary pre-allocated NumPy array to store the shortest distance between atoms in different nodes.

- **contact_mat** (*Any*) – Pre-allocated NumPy matrix where node contacts will be stored.
- **atom_to_node** (*Any*) – NumPy array that maps atoms in atom groups to their respective nodes.
- **node_group_indices_np** (*Any*) – NumPy array with atom indices for all atoms in each node group.
- **node_group_indices_np_aux** (*Any*) – Auxiliary NumPy array with the indices of the first atom in each atom group, as listed in *node_group_indices_np*.

Return type
None

dynetan.contact.**dist_to_contact_par**(*num_nodes*, *n_atoms*, *cutoff_dist*, *tmp_dists*, *contact_mat*, *atom_to_node*, *node_group_indices_np*, *node_group_indices_np_aux*, *n_cores=1*)

(PARALlEL) Translates MDAnalysis distance calculation to node contact matrix.

This function is a parallel version of *dist_to_contact()*. It prepares the calculations and initialized *n_cores* processes using the *dist_to_contact_proc()* function.

Parameters

- **num_nodes** (*int*) – Number of nodes in the system.
- **n_atoms** (*int*) – Number of atoms in atom groups represented by system nodes. Usually hydrogen atoms are not included in contact detection, and are not present in atom groups.
- **cutoff_dist** (*float*) – Distance at which atoms are no longer considered 'in contact'.
- **tmp_dists** (*Any*) – Temporary pre-allocated NumPy array with atom distances. This is the result of MDAnalysis *self_distance_array* calculation.
- **contact_mat** (*Any*) – Pre-allocated NumPy matrix where node contacts will be stored.
- **atom_to_node** (*Any*) – NumPy array that maps atoms in atom groups to their respective nodes.
- **node_group_indices_np** (*Any*) – NumPy array with atom indices for all atoms in each node group.
- **node_group_indices_np_aux** (*Any*) – Auxiliary NumPy array with the indices of the first atom in each atom group, as listed in *node_group_indices_np*.
- **n_cores** (*int*) – Number of cores used to parallelize calculation.

Return type
None

dynetan.contact.**dist_to_contact_proc**(*n_atoms*, *cutoff_dist*, *tmp_dists*, *atom_to_node*, *node_group_indices_np*, *node_group_indices_np_aux*, *in_queue*, *out_queue*)

(PROCESS) Translates MDAnalysis distance calculation to node contact matrix.

This function is a parallel version of *dist_to_contact()*. It executes the calculations prepared by the *dist_to_contact_par()* function.

Parameters

- **n_atoms** (*int*) – Number of atoms in atom groups represented by system nodes. Usually hydrogen atoms are not included in contact detection, and are not present in atom groups.
- **cutoff_dist** (*float*) – Distance at which atoms are no longer considered 'in contact'.

- **tmp_dists** (*Any*) – Temporary pre-allocated NumPy array with atom distances. This is the result of MDAnalysis *self_distance_array* calculation.
- **atom_to_node** (*Any*) – NumPy array that maps atoms in atom groups to their respective nodes.
- **node_group_indices_np** (*Any*) – NumPy array with atom indices for all atoms in each node group.
- **node_group_indices_np_aux** (*Any*) – Auxiliary NumPy array with the indices of the first atom in each atom group, as listed in *node_group_indices_np*.
- **in_queue** (*Any*) – Multiprocessing queue object for acquiring jobs.
- **out_queue** (*Any*) – Multiprocessing queue object for placing results.

Return type
None

dynetan.contact.**get_contacts_c**(*selection*, *num_nodes*, *cutoff_dist*, *tmp_dists*, *tmp_dists_atms*, *contact_mat*, *atom_to_node*, *node_group_indices_np*, *node_group_indices_np_aux*, *dist_mode=0*, *ncores=1*)

Executes MDAnalysis atom distance calculation and node contact detection.

This function is JIT compiled with Numba as a wrapper for two optimized distance calculation and contact determination calls. The first is MDAnalysis' *self_distance_array*. The second is the internal *dist_to_contact()*. All results are stored in pre-allocated NumPy arrays.

Parameters

- **selection** (*MDAnalysis.AtomGroup*) – Atom selection for the system being analyzed.
- **num_nodes** (*int*) – Number of nodes in the system.
- **cutoff_dist** (*float*) – Distance at which atoms are no longer considered 'in contact'.
- **tmp_dists** (*Any*) – Temporary pre-allocated NumPy array with atom distances. This is the result of MDAnalysis *self_distance_array* calculation.
- **tmp_dists_atms** (*Any*) – Temporary pre-allocated NumPy array to store the shortest distance between atoms in different nodes.
- **contact_mat** (*Any*) – Pre-allocated NumPy matrix where node contacts will be stored.
- **atom_to_node** (*Any*) – NumPy array that maps atoms in atom groups to their respective nodes.
- **node_group_indices_np** (*Any*) – NumPy array with atom indices for all atoms in each node group.
- **node_group_indices_np_aux** (*Any*) – Auxiliary NumPy array with the indices of the first atom in each atom group, as listed in *node_group_indices_np*.
- **dist_mode** (*int*) – Method for distance calculation in MDAnalysis (all or capped).
- **ncores** (*int*) – Number of cores used to process cartesian distance between network nodes.

Return type
None

dynetan.contact.**get_lin_index_numba**(*src*, *trgt*, *n*)

Conversion from 2D matrix indices to 1D triangular.

Converts from 2D matrix indices to 1D (n*(n-1)/2) unwrapped triangular matrix index. This function is JIT compiled using Numba.

Parameters

- **src** (*int*) – Source node.
- **trgt** (*int*) – Target node.
- **n** (*int*) – Dimension of square matrix

Returns
1D index in unwrapped triangular matrix.

Return type
int

6.4 Generalized Correlations

This module contains auxiliary functions for the parallel calculation of generalized correlation coefficients.

dynetan.gencor.**calc_cor_proc**(*traj*, *win_len*, *psi*, *phi*, *num_dims*, *k_neighb*, *in_queue*, *out_queue*)

Process for parallel calculation of generalized correlation coefficients.

This function serves as a wrapper and manager for the calculation of generalized correlation coefficients. It uses Python's *multiprocessing* module to launch *Processes* for parallel execution, where each process uses two *multiprocessing* queues to manage job acquisition and saving results. For each calculation, the job acquisition queue passes a trajectories of a pair of atoms, while the output queue stores the generalized correlation coefficient. The generalized correlation coefficient is calculated using a mutual information coefficient which is estimated using an optimized function.

Note: Please refer to *calc_mir_numba_2var()* for details about the calculation of mutual information coefficients.

Parameters

- **traj** (*Any*) – NumPy array with trajectory information.
- **win_len** (*int*) – Number of trajectory frames in the current window.
- **psi** (*Any*) – Pre-calculated parameter used for mutual information estimation.
- **phi** (*Any*) – Pre-calculated parameter used for mutual information estimation.
- **num_dims** (*int*) – Number of dimensions in trajectory data (usually 3 dimensions, for X,Y,Z coordinates).
- **k_neighb** (*int*) – Parameter used for mutual information estimation.
- **in_queue** (*Any*) – Multiprocessing queue object for acquiring jobs.
- **out_queue** (*Any*) – Multiprocessing queue object for placing results.

Return type
None

dynetan.gencor.**calc_mir_numba_2var**(*traj*, *num_frames*, *num_dims*, *k_neighb*, *psi*, *phi*)

Calculate mutual information coefficients.

This function estimates the mutual information coefficient based on Kraskov et. al. (2004), using the rectangle method. This implementation is hardcoded for 2 variables, to maximize efficiency, and is Just-In-Time (JIT) compiled using Numba.

This calculation assumes that the input trajectory of two random variables (in this case, positions of two atoms) is provided in a variable-dimension-step format (or "atom by x/y/z by frame" for molecular dynamics data). It also assumes that all trajectory data has been standardized (see *stand_vars_c()*).

Parameters

- **traj** (*Any*) – NumPy array with trajectory information.
- **num_frames** (*int*) – Number of trajectory frames in the current window.
- **num_dims** (*int*) – Number of dimensions in trajectory data (usually 3 dimensions, for X,Y,Z coordinates).
- **k_neighb** (*int*) – Parameter used for mutual information estimation.
- **psi** (*numpy.ndarray*) – Pre-calculated parameter used for mutual information estimation.
- **phi** (*numpy.ndarray*) – Pre-calculated parameter used for mutual information estimation.

Return type
float

dynetan.gencor.**mir_to_corr**(*mir*, *num_dims=3*)

Transforms Mutual Information R into Generalized Correlation Coefficient

Returns
Generalized Correlation Coefficient (float)

Parameters

- **mir** (*float*) –
- **num_dims** (*int*) –

Return type
float

dynetan.gencor.**prep_mi_c**(*universe*, *traj*, *beg*, *end*, *num_nodes*, *num_dims*)

Standardize variables in trajectory data.

This function stores the trajectory data in a new format to accelerate the estimation of mutual information coefficients. The estimation of mutual information coefficients assumes that the input trajectory of random variables (in this case, positions of atoms) is provided in a variable-dimension-step format (or "atom by x/y/z by frame" for molecular dynamics data). However, the standard MDAnalysis trajectory format if "frame by atom by x/y/z".

This function also calls a dedicated function to standardize the atom position data, also necessary for mutual information estimation.

Note: Please refer to *stand_vars_c()* for details about the data standardization process. Please refer to *calc_mir_numba_2var()* for details about the calculation of mutual information coefficients.

Parameters

- **universe** (*Any*) – MDAnalysis universe object containing all trajectory information.
- **traj** (*Any*) – NumPy array where trajectory information will be stored.
- **beg** (*int*) – Initial trajectory frame to be used for analysis.
- **end** (*int*) – Final trajectory frame to be used for analysis.
- **num_nodes** (*int*) – Number of nodes in the network.

- **num_dims** (*int*) – Number of dimensions in trajectory data (usually 3 dimensions, for X,Y,Z coordinates).

Return type
None

dynetan.gencor.**stand_vars_c**(*traj*, *num_nodes*, *num_dims*)

Standardize variables in trajectory data.

This function prepares the trajectory for the estimation of mutual information coefficients. This calculation assumes that the input trajectory of random variables (in this case, positions of atoms) is provided in a variable-dimension-step format (or "atom by x/y/z by frame" for molecular dynamics data).

Note: Please refer to *prep_mi_c()* for details about the data conversion process. Please refer to *calc_mir_numba_2var()* for details about the calculation of mutual information coefficients.

Parameters

- **traj** (*Any*) – NumPy array with trajectory information.
- **num_nodes** (*int*) – Number of nodes in the network.
- **num_dims** (*int*) – Number of dimensions in trajectory data (usually 3 dimensions, for X,Y,Z coordinates).

Return type
None

6.5 Network Properties

This module contains auxiliary functions for the parallel calculation of network properties.

dynetan.network.**calcBetweenPar**(*nx_graphs*, *in_queue*, *out_queue*)

Wrapper to calculate betweenness in parallel.

The betweenness calculations used here only take into account the number of paths passing through a given edge, so no weight are considered.

For every window, the function stores in the output queue an ordered dict of node pairs with betweenness higher than zero.

Parameters

- **nx_graphs** (*Any*) – NetworkX graph object.
- **in_queue** (*Queue*) – Multiprocessing queue object for acquiring jobs.
- **out_queue** (*Queue*) – Multiprocessing queue object for placing results.

Return type
None

dynetan.network.**calcOptPathPar**(*nx_graphs*, *in_queue*, *out_queue*)

Wrapper to calculate Floyd Warshall optimal path in parallel.

For the FW optimal path determination, we use the node "distance" as weights, that is, the log-transformation of the correlations, NOT the correlation itself.

For every window, the function stores in the output queue the optimal paths. It turns the dictionary of distances returned by NetworkX into a NumPy 2D array per window of trajectory, which allows significant speed up in data analysis.

Parameters

- **nx_graphs** (*Any*) – NetworkX graph object.
- **in_queue** (*Queue*) – Multiprocessing queue object for acquiring jobs.
- **out_queue** (*Queue*) – Multiprocessing queue object for placing results.

Return type
None

6.6 Toolkit

This module contains auxiliary functions for manipulation of atom selections, acquiring pre-calculated cartesian distances, and user interface in jupyter notebooks.

dynetan.toolkit.**diagnostic**()

Diagnostic for parallelization of MDAnalysis.

Convenience function to detect if the current MDAnalysis installation supports OpenMP.

Return type
bool

dynetan.toolkit.**formatNodeGroups**(*atmGrp*, *nodeAtmStrL*, *grpAtmStrL=None*)

Format code to facilitate the definition of node groups.

This convenience function helps with the definition of node groups. It will produce formatted python code that the user can copy directly into a definition of atom groups.

If the entire residue is to be represented by a single node, then *grpAtmStrL* does not need to be defined. However, if more than one node is defined in *nodeAtmStrL*, then the same number of lists need to be added to *grpAtmStrL* to define each node group.

Parameters

- **atmGrp** (*Any*) – MDAnalysis atom group object with one residue.
- **nodeAtmStrL** (*list*) – Strings defining atoms that will represent nodes.
- **grpAtmStrL** (*list | None*) – Lists containing atoms that belong to each node group. If None, all atoms in the residue will be added to the same node group. This parameter can only be None when a single node is provided.

Returns

—

Return type
None

dynetan.toolkit.**getCartDist**(*src*, *trgt*, *numNodes*, *nodeDists*, *distype=0*)

Get cartesian distance between nodes.

Retrieves the cartesian distance between atoms representing nodes *src* and *trgt*. The *distype* argument causes the function to return the mean distance (type 0: default), Standard Error of the Mean (SEM) (type 1), minimum distance (type 2), or maximum distance (type 3).

Parameters

- **src** (*int*) – Source node.
- **trgt** (*int*) – Target node.
- **numNodes** (*int*) – Total number of nodes in the system.
- **distype** (*int*) – Type of cartesian distance output.
- **nodeDists** (*numpy.ndarray*) –

Returns
One of four types of measurements regarding the cartesian distance between nodes. See description above.

Return type
float

dynetan.toolkit.**getLinIndexC**(*src*, *trgt*, *dim*)

Conversion from 2D matrix indices to 1D triangular.

Converts from 2D matrix indices to 1D (n*(n-1)/2) unwrapped triangular matrix index.

Parameters

- **src** (*int*) – Source node.
- **trgt** (*int*) – Target node.
- **dim** (*int*) – Dimension of square matrix

Returns
1D index in unwrapped triangular matrix.

Return type
int

dynetan.toolkit.**getNGLSelFromNode**(*nodeIndx*, *atomsel*, *atom=True*)

Creates an atom selection for NGLView.

Returns an atom selection for a whole residue or single atom in the NGL syntax.

Parameters

- **nodeIndx** (*int*) – Index of network node.
- **atomsel** (*Any*) – MDAnalysis atom-selection object.
- **atom** (*bool*) – Determines if the selection should cover the entire residue, or just the representative atom.

Returns
Text string with NGL-style atom selection.

Return type
str

dynetan.toolkit.**getNodeFromSel**(*selection*, *atmsel*, *atm_to_node*)

Gets the node index from an atom selection.

Returns one or more node indices when given an MDAnalysis atom selection string.

Parameters

- **selection** (*str*) – MDAnalysis atom selection string.

- **atmsel** (*Any*) – MDAnalysis atom-selection object.
- **atm_to_node** (*Any*) – Dynamic Network Analysis atom-to-node mapping object.

Returns
List of node indices mapped to the provided atom selection.

Return type
numpy.ndarray

dynetan.toolkit.**getPath**(*src*, *trg*, *nodesAtmSel*, *preds*, *win=0*)

Gets connecting path between nodes.

This function recovers the list of nodes that connect *src* and *trg* nodes. An internal sanity check is performed to see if both nodes belong to the same residue. This may be the case in nucleic acids, for example, where two nodes are used to describe the entire residue.

Parameters

- **src** (*int*) – Source node.
- **trg** (*int*) – Target node.
- **nodesAtmSel** (*Any*) – MDAnalysis atom-selection object.
- **preds** (*dict*) – Predecessor data in dictionary format.
- **win** (*int*) – Selects the simulation window used to create optimal paths.

Returns

A NumPy array with the list of nodes or an empty list in case
no optimal path could be found.

Return type
numpy.ndarray

dynetan.toolkit.**getSelFromNode**(*nodeIndx*, *atomsel*, *atom=False*)

Gets the MDanalysis selection string from a node index.

Given a node index, this function builds an atom selection string in the following format: resname and resid and segid [and name]

Parameters

- **nodeIndx** (*int*) – Index of network node.
- **atomsel** (*Any*) – MDAnalysis atom-selection object.
- **atom** (*bool*) – Determines if the selection should cover the entire residue, or just the representative atom.

Returns
Text string with MDAnalysis-style atom selection.

Return type
str

dynetan.toolkit.**showNodeGroups**(*nv_view*, *atm_grp*, *usr_node_groups*, *node_atm_sel=''*)

Labels atoms in an NGLview instance to visualize node groups.

This convenience function helps with the definition of node groups. It will label atoms and nodes in a structure to help visualize the selection of atoms and nodes.

Parameters

- **nv_view** (*Any*) – The initialized NGLview object.
- **atm_grp** (*Any*) – The MDanalysis atom group object containing one residue.
- **usr_node_groups** (*dict*) – A dictionary of dictionaries with node groups for a given residue.
- **node_atm_sel** (*str*) – A string selecting a node atom so that only atoms in that group are labeled.

Returns

—

Return type
None

6.7 Visualization

This module contains auxiliary functions for visualization of the system and network analysis results.

dynetan.viz.**getCommunityColors**()

Gets standardized colors for communities.

This function loads pre-specified colors that match those available in VMD.

Returns

Returns a pandas dataframe with a VMD-compatible color scale for node
communities.

Return type
pandas.DataFrame

dynetan.viz.**prepTclViz**(*base_name*, *num_winds*, *ligand_segid='NULL'*, *trg_dir='./'*)

Prepares system-specific TCL script vor visualization.

This function prepares a TCL script that can be loaded by VMD to create high-resolution renderings of the system.

Parameters

- **base_name** (*str*) – Base name for TCL script and data files necessary for visualization.
- **num_winds** (*int*) – Number of windows created for analysis.
- **ligand_segid** (*str*) – Segment ID of ligand residue. This will create a special representations for small molecules.
- **trg_dir** (*str*) – Directory where TCL and data files will be saved.

Return type
None

dynetan.viz.**showCommunityByID**(*nvView*, *cDF*, *clusID*, *system*, *refWindow*, *shapeCounter*, *nodesAtmSel*, *colorValDictRGB*, *trg_system*, *trg_window*)

Creates NGLView representation of nodes in a community.

Renders a series of spheres to represent all nodes in the selected community.

The *system* argument selects one dataset to ve used for the creation of standardized representations, which allows comparisons between variants of the same system, with mutations or different ligands, for example.

The *colorValDictRGB* argument relates colors to RGB codes. This allows the creation of several independent representations using the same color scheme.

For examples of formatted data, see Dynamical Network Analysis tutorial.

Parameters

- **nvView** (*Any*) – NGLView object.
- **cDF** (*Any*) – Pandas data frame relating node IDs with their communities in every analyzed simulation window. This requires a melt format.
- **clusID** (*float*) – ID of the community (or cluster) to be rendered.
- **system** (*str*) – System used as reference to standardize representation.
- **refWindow** (*int*) – Window used as reference to standardize representation.
- **shapeCounter** (*Any*) – Auxiliary list to manipulate NGLView representations.
- **nodesAtmSel** (*Any*) – MDAnalysis atom selection object.
- **colorValDictRGB** (*Any*) – Dictionary that standardizes community colors.
- **trg_system** (*str*) – System to be used for rendering.
- **trg_window** (*int*) – Window used for rendering.

Return type
None

dynetan.viz.**showCommunityByNodes**(*nvView*, *cDF*, *nodeList*, *system*, *refWindow*, *shapeCounter*, *nodesAtmSel*, *colorValDictRGB*)

Creates NGLView representation of nodes in a community.

Renders a series of spheres to represent all nodes in the selected community.

Parameters

- **nvView** (*Any*) – NGLView object.
- **cDF** (*Any*) – Pandas data frame relating node IDs with their communities in every analyzed simulation window. This requires a melt format.
- **nodeList** (*list*) – List of node IDs to be rendered.
- **system** (*str*) – System used as reference to standardize representation.
- **refWindow** (*int*) – Window used as reference to standardize representation.
- **shapeCounter** (*Any*) – Auxiliary list to manipulate NGLView representations.
- **nodesAtmSel** (*Any*) – MDAnalysis atom selection object.
- **colorValDictRGB** (*Any*) – Dictionary that standardizes community colors.

Return type
None

dynetan.viz.**showCommunityByTarget**(*nvView*, *nodeCommDF*, *trgtNodes*, *window*, *nodesAtmSel*, *dnad*, *colorValDict*)

Creates NGLView representation of edges between selected nodes and their contacts.

Renders a series of cylinders to represent all edges that connect selected nodes with other nodes in contact. Only nodes that have been assigned to a community are shown, to minimize the occurrence of unstable contacts. Edges between nodes in different communities are still rendered, but shown in different representations.

Parameters

- **nvView** (*Any*) – NGLView object.
- **nodeCommDF** (*Any*) – Pandas data frame relating node IDs with their communities.
- **trgtNodes** (*list*) – List of node IDs.
- **window** (*int*) – Window used for representation.
- **nodesAtmSel** (*Any*) – MDAnalysis atom selection object.
- **dnad** (*Any*) – Dynamical Network Analysis data object.
- **colorValDict** (*Any*) – Dictionary that standardizes community colors.

Return type
None

dynetan.viz.**showCommunityGlobal**(*nvView*, *nodeCommDF*, *commID*, *window*, *nodesAtmSel*, *dnad*, *colorValDict*)

Creates NGLView representation of a specified community.

Renders a series of cylinders to represent all edges in the network that connect nodes in the same community. Edges between nodes in different communities are not rendered.

Parameters

- **nvView** (*Any*) – NGLView object.
- **nodeCommDF** (*Any*) – Pandas data frame relating node IDs with their communities.
- **commID** (*float*) – Community ID for the community to be rendered.
- **window** (*int*) – Window used for representation.
- **nodesAtmSel** (*Any*) – MDAnalysis atom selection object.
- **dnad** (*Any*) – Dynamical Network Analysis data object.
- **colorValDict** (*Any*) – Dictionary that standardizes community colors.

Return type
None

dynetan.viz.**viewPath**(*nvView*, *path*, *dists*, *maxDirectDist*, *nodesAtmSel*, *win=0*, *opacity=0.75*, *color='green'*, *side='both'*, *segments=5*, *disableImpostor=True*, *useCylinder=True*)

Creates NGLView representation of a path.

Renders a series of cylinders to represent a network path. The maxDirectDist argument is used as a normalization factor to scale representations of edges. For more details on NGL parameters, see NGLView's documentation.

Parameters

- **nvView** (*Any*) – NGLView object.
- **path** (*list*) – Sequence of nodes that define a path.
- **maxDirectDist** (*float*) – Maximum direct distance between nodes in network.
- **nodesAtmSel** (*Any*) – MDAnalysis atom selection object.
- **win** (*int*) – Window used for representation.
- **opacity** (*float*) – Controls edge opacity.
- **color** (*str*) – Controls edge color.

- **side** (*str*) – Controls edge rendering quality.
- **segments** (*int*) – Controls edge rendering quality.
- **disableImpostor** (*bool*) – Controls edge rendering quality.
- **useCylinder** (*bool*) – Controls edge rendering quality.
- **dists** (*Any*) –

Return type
None

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

7.1 Resources and References

7.1.1 Citing

To cite this package please use the following publication:

- Generalized correlation-based dynamical network analysis: a new high-performance approach for identifying allosteric communications in molecular dynamics trajectories. JCP (2020). DOI: 10.1063/5.0018980

For further discussion and scientific background, please refer to:

- Experimental and computational determination of tRNA dynamics. FEBS Letters (2010). DOI: 10.1016/j.febslet.2009.11.061
- Exit strategies for charged tRNA from GluRS. JMB (2010). DOI: 10.1016/j.jmb.2010.02.003
- Dynamical Networks in tRNA:protein complexes. PNAS (2009). DOI: 10.1073/pnas.0810961106

7.2 Last Updated

Date
Aug 23, 2023

PYTHON MODULE INDEX

d

INDEX

M

P

S

V