menten_gcn

Release 0.4.0

Menten AI, Inc.

May 27, 2021

CONTENTS

1 Contents **3**

1.1 Overview . . . 3
1.1.1 Graph Layout . . . 3
1.1.2 Graph Tensors . . . 4
1.1.3 Usage . . . 5
1.2 Decorator Menu . . . 6
1.2.1 Geometry . . . 6
1.2.2 Sequence . . . 8
1.2.3 Rosetta . . . 8
1.3 Classes . . . 12
1.3.1 DataMaker . . . 13
1.3.2 Decorators . . . 16
1.3.3 Pose Wrappers . . . 29
1.4 Examples . . . 29
1.4.1 Hello World . . . 29
1.4.2 Simple Train . . . 31
1.4.3 Sparse Mode . . . 32
1.4.4 Custom Decorator . . . 34
1.5 Technical Overview . . . 37
1.5.1 Documentation . . . 37
1.5.2 Installation . . . 37
1.5.3 Development . . . 38
1.6 Troubleshooting . . . 38
1.6.1 Sparse Mode . . . 38
1.6.2 Versioning . . . 38
1.7 Authors . . . 38
1.8 Indices and tables . . . 38

Index **39**

At the command line:

```
pip install menten-gcn
```

CHAPTER

ONE

CONTENTS

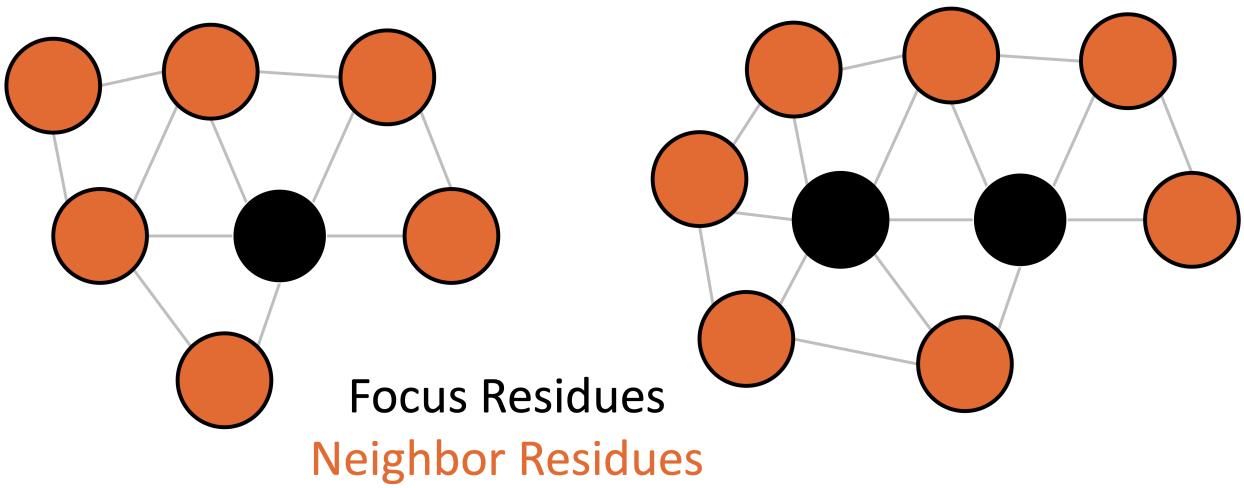
1.1 Overview

The goal of Menten GCN is to create GCN tensors from protein models (poses). We are aligning with Spektral's vocabulary style when talking about GCNs and Rosetta's vocabulary when talking about poses.

1.1.1 Graph Layout

Each node (vertex) in our graph represents a single residue position. Edges connect nodes that are close in 3D space. Our goal in Menten GCN is to analyze small pockets of residues at a time, though the size of each pocket is entirely up to the user and can encompass the entire protein if you wish.

We generate a graph by first declaring one or more "focus" residues. These residues will be at the center of our pocket. Menten GCN will automatically select the residue positions closest in space to the focus residues and will use them to build neighbor nodes. Menten GCN will also automatically add edges between any two nodes that are close in space.



1.1.2 Graph Tensors

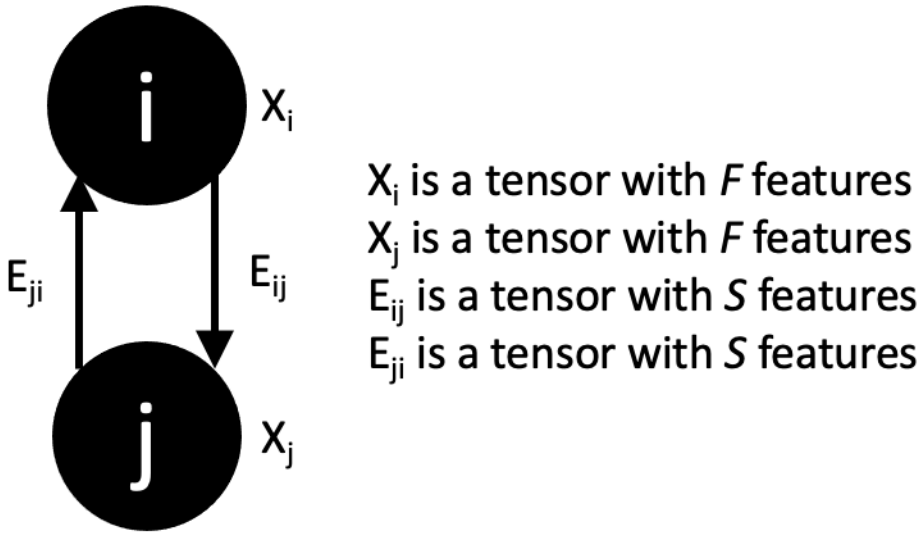
We have 3 primary parameters in this system:

- "N" is maximum the number of nodes in any graph. This includes focus nodes and neighbor nodes
- "F" is the number of features per node
- "S" is the number of features per edge

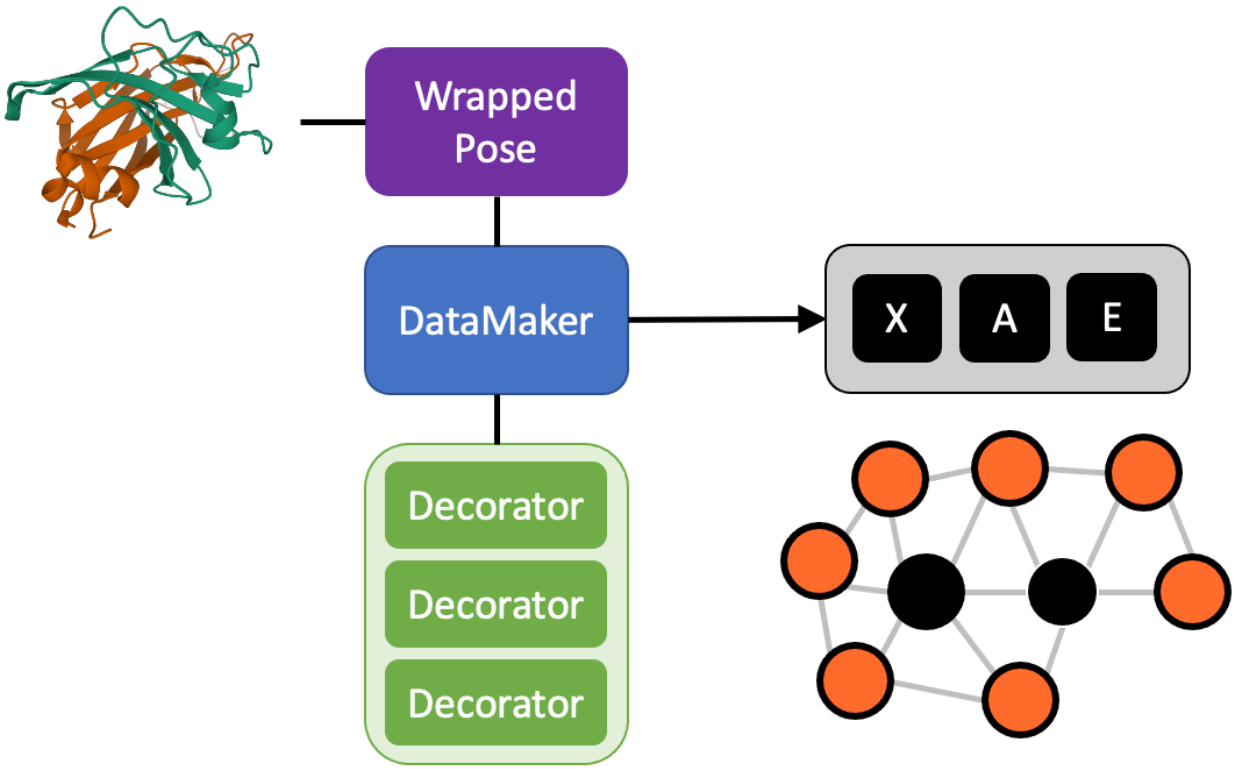
These parameters are used to define 3 input tensors:

- Tensor "X" holds the node features and is of shape (N,F)
- Tensor "A" holds the adjacency matrix and is of shape (N,N)
- Tensor "E" holds the edge features and is of shape (N,N,S)

One nuance of the "E" tensor is that edges can have direction. Every pair of residues has room for two edge tensors in our system. Some of our edge features are symmetric (like distance) so they will have the same value going in both directions. Other edge tensors are asymmetric (like relative geometries) so they will have different values for each of the two slots in "E".



1.1.3 Usage



1. Start by loading your pose in python using any of our supported packages.
    - Just Rosetta and MDTraj right now. Get in touch if you want more!
2. Wrap your pose using the appropriate wrapper for your package.
    - See Classes -> Pose Wrappers
3. Define a list of decorators to use to represent your pose.
    - See Classes -> Decorators
    - An example decorator would be PhiPsiRadians, which decorates each node with its Phi and Psi value
4. Use this list of decorators to build a DataMaker
5. The DataMaker will then take your wrapped pose, ask for the focus residues, and return the X, A, and E tensors
6. From here you have a few choices.
    - You can train on these tensors directly
    - You can utilize Spektral's Dataset interface to make training easier with large amounts of data
    - Or you can save these for later. Stick them on disk and come back to them when you're ready to train

See the DataMaker class and examples for more details.

1.2 Decorator Menu

1.2.1 Geometry

class menten_gcn.decorators.**CACA_dist**(*use_nm: bool = False*)
Measures distance between the two C-Alpha atoms of each residue

- 0 Node Features
- 1 Edge Feature

Parameters **use_nm** (*bool*) – If true (default), measure distance in Angstroms. Otherwise use nanometers.

class menten_gcn.decorators.**CBCB_dist**(*use_nm: bool = False*)
Measures distance between the two C-Beta atoms of each residue. Note: We will calculate the "ideal ALA" CB location even if this residue has a CB atom. This may sound silly but it is intended to prevents noise from different native amino acid types.

- 0 Node Features
- 1 Edge Feature

Parameters **use_nm** (*bool*) – If true (default), measure distance in Angstroms. Otherwise use nanometers.

class menten_gcn.decorators.**PhiPsiRadians**(*sincos: bool = False*)
Returns the phi and psi values of each residue position.

- 2-4 Node Features
- 0 Edge Features

Parameters **sincos** (*bool*) – Return the sine and cosine of phi and psi instead of just the raw values.

class menten_gcn.decorators.**ChiAngleDecorator**(*chi1: bool = True*, *chi2: bool = True*, *chi3: bool = True*, *chi4: bool = True*, *sincos: bool = True*)
Returns the chi values of each residue position. Ranges from -pi to pi or -1 to 1 if sincos=True.

WARNING: This can behave inconsistantly for proton chis accross modeling frameworks. Rosetta adds hydrogens when they are absent from the input file but MDtraj does not. This results in Rosetta calculating a chi value in some cases that MDtraj skips!

- 0-8 Node Features
- 0 Edge Features

Parameters

- **chi1** (*bool*) – Include chi1's value
- **chi2** (*bool*) – Include chi2's value
- **chi3** (*bool*) – Include chi3's value
- **chi4** (*bool*) – Include chi4's value
- **sincos** (*bool*) – Return the sine and cosine of chi instead of just the raw values

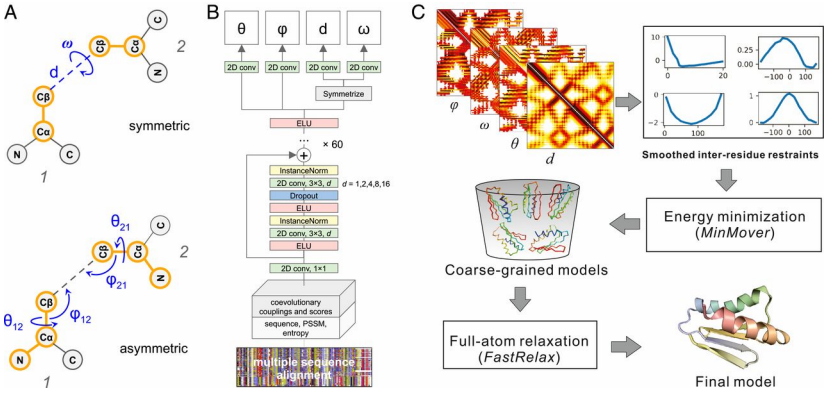
class menten_gcn.decorators.**trRosettaEdges**(*sincos: bool = False*, *use_nm: bool = False*)
Use the residue pair geometries used in this paper: https://www.pnas.org/content/117/3/1496/tab-figures-data

- 0 Node Features
- 4-7 Edge Features

Parameters

- **sincos** (*bool*) – Return the sine and cosine of phi and psi instead of just the raw values.
- **use_nm** (*bool*) – If true, measure distance in Angstroms. Otherwise use nanometers.

  Note: This default value does not match the default of other decorators. This is for the sake of matching the trRosetta paper.



class menten_gcn.decorators.**SimpleBBGeometry**(*use_nm=False*)
Meta-decorator that combines PhiPsiRadians(sincos=False) and CBCB_dist

- 2 Node Features
- 1 Edge Feature

Parameters **use_nm** (*bool*) – If true, measure distance in Angstroms. Otherwise use nanometers.

class menten_gcn.decorators.**StandardBBGeometry**(*use_nm=False*)
Meta-decorator that combines PhiPsiRadians(sincos=True) and trRosettaEdges(sincos=False)

- 4 Node Features
- 4 Edge Features

Parameters **use_nm** (*bool*) – If true, measure distance in Angstroms. Otherwise use nanometers.

class menten_gcn.decorators.**AdvancedBBGeometry**(*use_nm=False*)
Meta-decorator that combines PhiPsiRadians(sincos=True), CACA_dist, and trRosettaEdges(sincos=True)

- 4 Node Features
- 8 Edge Features

Parameters **use_nm** (*bool*) – If true, measure all distances in Angstroms. Otherwise use nanometers.

1.2.2 Sequence

class menten_gcn.decorators.**Sequence**
One-hot encode the canonical amino acid identity on each node.

- 20 Node Features
- 0 Edge Features

class menten_gcn.decorators.**DesignableSequence**
One-hot encode the canonical amino acid identity on each node, with a 21st value for residues that are not yet assigned an amino acid identity.

Note: requires you to call WrappedPose.set_designable_resids first

- 21 Node Features
- 0 Edge Features

class menten_gcn.decorators.**SequenceSeparation**(*ln: bool = True*)
The sequence distance between the two residues (i.e., number of residues between these two residues in sequence space, plus one). -1.0 if the two residues belong to different chains.

- 0 Node Features
- 1 Edge Feature

Parameters **ln** (*bool*) – Report the natural log of the distance instead of the raw count. Does not apply to -1 values

class menten_gcn.decorators.**SameChain**
1 if the two residues are part of the same protein chain. Otherwise 0.

- 0 Node Features
- 1 Edge Feature

1.2.3 Rosetta

class menten_gcn.decorators.**RosettaResidueSelectorDecorator**(*selector*, *description: str*)
Takes a user-provided residue selctor and labels each residue with a 1 or 0 accordingly.

- 1 Node Feature
- 0 Edge Features

Parameters

- **selector** (*ResidueSelector*) – This residue selector will be applied to the Rosetta pose
- **description** (*str*) – This is the string that will label this feature in the final summary. Not technically required but highly recommended

Example:

```
import menten_gcn as mg
import menten_gcn.decorators as decs
import pyrosetta

pyrosetta.init()
```

(continues on next page)

(continued from previous page)

```
buried = pyrosetta.rosetta.core.select.residue_selector.LayerSelector()
buried.set_layers( True, False, False )
buried_dec = decs.RosettaResidueSelectorDecorator( selector=buried, description='
↪<Layer select_core="true" />' )

data_maker = mg.DataMaker( decorators=[ buried_dec ], edge_distance_cutoff_A=10.0,␣
↪max_residues=30 )
data_maker.summary()
```

Gives:

```
Summary:

2 Node Features:
1 : 1 if the node is a focus residue, 0 otherwise
2 : 1.0 if the residue is selected by the residue selector, 0.0 otherwise. User␣
↪defined definition of the residue selector and how to reproduce it: <Layer select_
↪core="true" />

1 Edge Features:
1 : 1.0 if the two residues are polymer-bonded, 0.0 otherwise
```

Note that the additional features are due to the BareBonesDecorator, which is included by default

class menten_gcn.decorators.**RosettaResidueSelectorFromXML**(*xml_str: str*, *res_sele_name: str*)

Takes a user-provided residue selctor via XML and labels each residue with a 1 or 0 accordingly.

- 1 Node Feature
- 0 Edge Features

Parameters

- **xml_str** (*str*) – XML snippet that defines the selector
- **res_sele_name** (*str*) – The name of the selector within the snippet

Example:

```
import menten_gcn as mg
import menten_gcn.decorators as decs
import pyrosetta

pyrosetta.init()
xml = '''
<RESIDUE_SELECTORS>
<Layer name="surface" select_surface="true" />
</RESIDUE_SELECTORS>
'''
surface_dec = decs.RosettaResidueSelectorFromXML( xml, "surface" )

max_res=30
```

(continues on next page)

(continued from previous page)

```
data_maker = mg.DataMaker( decorators=[ surface_dec ], edge_distance_cutoff_A=10.0,␣
↪max_residues=max_res )
data_maker.summary()
```

Gives:

```
Summary:

2 Node Features:
1 : 1 if the node is a focus residue, 0 otherwise
2 : 1.0 if the residue is selected by the residue selector, 0.0 otherwise. User␣
↪defined definition of the residue selector and how to reproduce it: Took the␣
↪residue selector named surface from this XML:
<RESIDUE_SELECTORS>
<Layer name="surface" select_surface="true" />
</RESIDUE_SELECTORS>


1 Edge Features:
1 : 1.0 if the two residues are polymer-bonded, 0.0 otherwise
```

Note that the additional features are due to the BareBonesDecorator, which is included by default

class menten_gcn.decorators.**RosettaJumpDecorator**(*use_nm: bool = False*, *rottype: str = 'euler'*)

Measures the translational and rotational relationships between all residue pairs. This uses internal coordinate frames so it is agnostic to the global coordinate system. You can move/rotate your protein around and these will stay the same.

- 0 Node Features
- 6-12 Edge Features

Parameters

- **use_nm** (*bool*) – If true (default), measure distance in Angstroms. Otherwise use nanometers.
- **rottype** (*str*) – How do you want to represent the rotational degrees of freedom? Options are "euler" (default), "euler_sincos", "matrix", "quat", "rotvec", and "rotvec_sincos".

class menten_gcn.decorators.**RosettaHBondDecorator**(*sfxn=None*, *bb_only: bool = False*)

Takes a user-provided residue selctor via XML and labels each residue with a 1 or 0 accordingly.

- 0 Node Features
- 1-5 Edge Features (depending on bb_only)

Parameters

- **sfxn** (*ScoreFunction*) – Score function used to calculate hbonds. We will use Rosetta's default if this is None
- **bb_only** (*bool*) – Only consider backbone-backbone hbonds. Reduces the number of features from 5 down to 1

class menten_gcn.decorators.**Rosetta_Ref2015_OneBodyEneriges**(*individual: bool = False*, *score_types=None*)

Label each node with its Rosetta one-body energy

- 1 - 20-ish Node Features
- 0 Edge Features

Parameters

- **individual** (*bool*) – If true, list the score for each term individually. Otherwise sum them all into one value.
- **score_types** (*list of ScoreTypes*) – Only use these score types. None (default) includes all default types. Note - this only applies if individual == True

class menten_gcn.decorators.**Rosetta_Ref2015_TwoBodyEneriges**(*individual: bool = False*, *score_types=None*)

Label each edge with its Rosetta two-body energy

- 0 Node Features
- 1 - 20-ish Edge Features

Parameters

- **individual** (*bool*) – If true, list the score for each term individually. Otherwise sum them all into one value.
- **score_types** (*list of ScoreTypes*) – Only use these score types. None (default) includes all default types. Note - this only applies if individual == True

class menten_gcn.decorators.**Ref2015Decorator**(*individual: bool = False*, *score_types=None*)

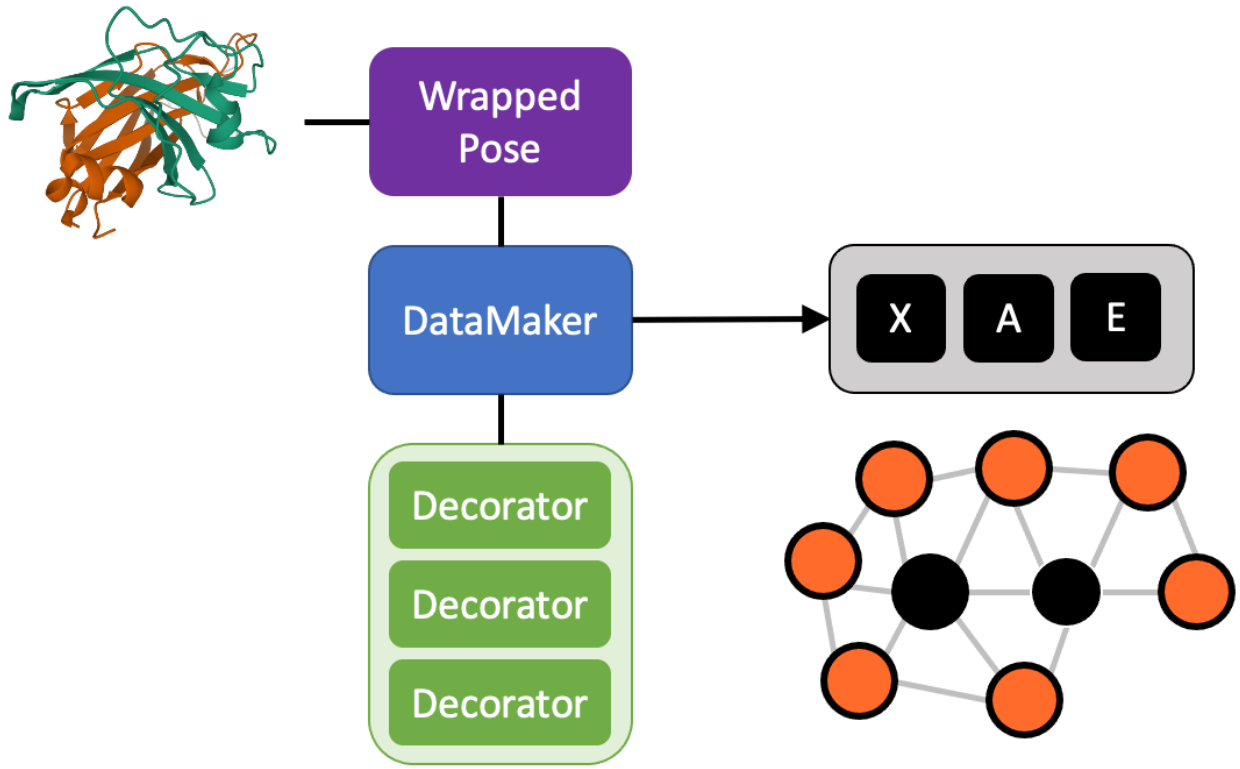
Meta-decorator that combines Rosetta_Ref2015_OneBodyEneriges and Rosetta_Ref2015_TwoBodyEneriges

- 1 - 20-ish Node Features
- 1 - 20-ish Edge Features

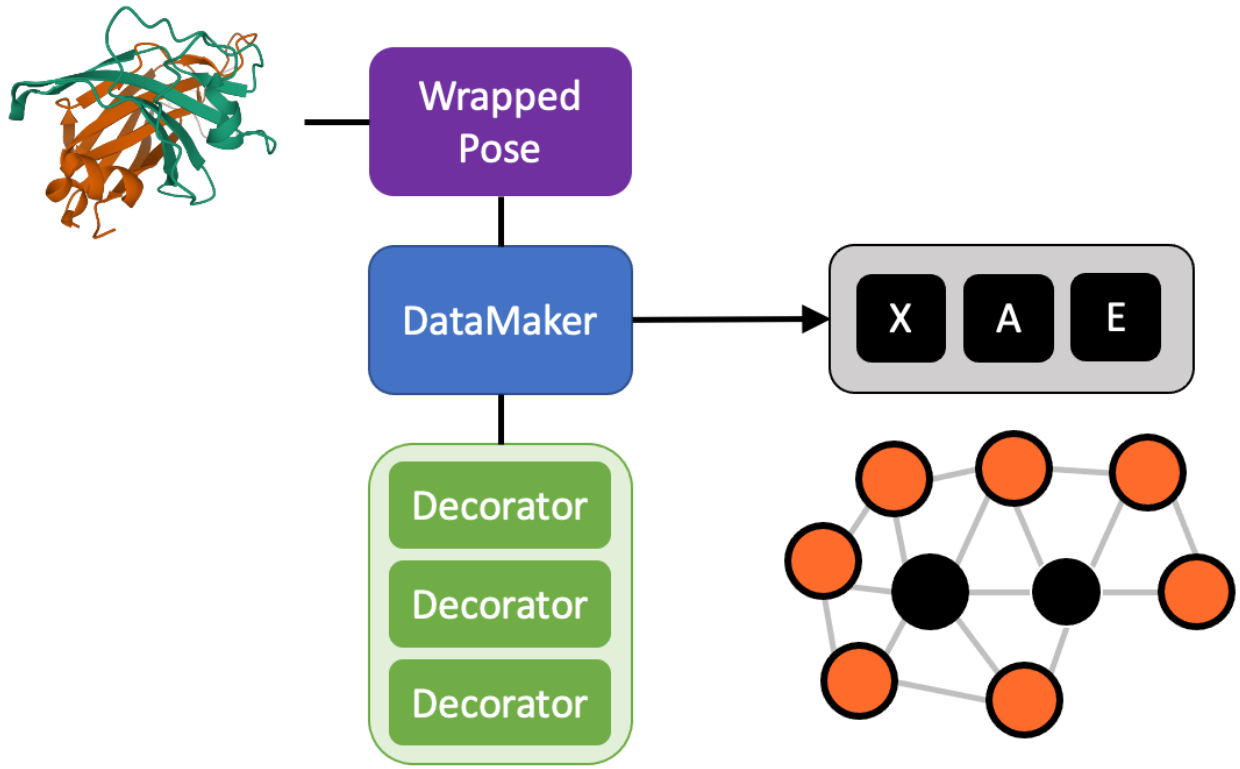
Parameters

- **individual** (*bool*) – If true, list the score for each term individually. Otherwise sum them all into one value.
- **score_types** (*list of ScoreTypes*) – Only use these score types. None (default) includes all default types. Note - this only applies if individual == True

1.3 Classes


Wrapped
Pose
DataMaker
X
A
E
Decorator
Decorator
Decorator

1.3.1 DataMaker



The DataMaker is the main character of Menten GCN. It has the job of applying decorators to poses and organizing them as tensors.

class menten_gcn.**DataMaker**(*decorators: List[menten_gcn.decorators.base.Decorator], edge_distance_cutoff_A: float, max_residues: int, exclude_bbdec: bool = False, nbr_distance_cutoff_A: Optional[float] = None, dtype: numpy.dtype = <class 'numpy.float32'>*)

The DataMaker is the user's interface for controlling the size and composition of their graph.

Parameters

- **decorators** (*list*) – List of decorators that you want to include
- **edge_distance_cutoff_A** (*float*) – An edge will be created between any two pairs of residues if their C-alpha atoms are within this distance (measured in Angstroms)
- **max_residues** (*int*) – What is the maximum number of nodes a graph can have? This includes focus and neighbor nodes. If the number of focus+neighbors exceeds this number, we will leave out the neighbors that are farthest away in 3D space.
- **exclude_bbdec** (*bool*) – Every DataMaker has a standard "bare bones" decorator that is prepended to the list of decorators you provide. Set this to false to remove it entirely.
- **nbr_distance_cutoff_A** (*float*) – A node will be included in the graph if it is within this distance (Angstroms) of any focus node. A value of None will set this equal to edge_distance_cutoff_A
- **dtype** (*np.dtype*) – What numpy data type should we use to represent your data?

summary()

Print a summary of the graph decorations to console. The goal of this summary is to describe every feature with enough detail to be able to be reproduced externally. This will also print any relevant citation information for individual decorators.

```
import menten_gcn as mg
import menten_gcn.decorators as decs

decorators=[ decs.SimpleBBGeometry(), decs.Sequence() ]
data_maker = mg.DataMaker( decorators=decorators, edge_distance_cutoff_A=10.0,␣
↪max_residues=15 )
data_maker.summary()
```

```
Summary:

23 Node Features:
1 : 1 if the node is a focus residue, 0 otherwise
2 : Phi of the given residue, measured in radians. Spans from -pi to pi
3 : Psi of the given residue, measured in radians. Spans from -pi to pi
4 : 1 if residue is A, 0 otherwise
5 : 1 if residue is C, 0 otherwise
6 : 1 if residue is D, 0 otherwise
7 : 1 if residue is E, 0 otherwise
8 : 1 if residue is F, 0 otherwise
9 : 1 if residue is G, 0 otherwise
10 : 1 if residue is H, 0 otherwise
11 : 1 if residue is I, 0 otherwise
12 : 1 if residue is K, 0 otherwise
13 : 1 if residue is L, 0 otherwise
14 : 1 if residue is M, 0 otherwise
15 : 1 if residue is N, 0 otherwise
16 : 1 if residue is P, 0 otherwise
17 : 1 if residue is Q, 0 otherwise
18 : 1 if residue is R, 0 otherwise
19 : 1 if residue is S, 0 otherwise
20 : 1 if residue is T, 0 otherwise
21 : 1 if residue is V, 0 otherwise
22 : 1 if residue is W, 0 otherwise
23 : 1 if residue is Y, 0 otherwise

2 Edge Features:
1 : 1.0 if the two residues are polymer-bonded, 0.0 otherwise
2 : Euclidean distance between the two CB atoms of each residue, measured in␣
↪Angstroms. In the case of GLY, use an estimate of ALA's CB position
```

get_N_F_S() → Tuple[int, int, int]

Returns

- **N** (*int*) – Maximum number of nodes in the graph
- **F** (*int*) – Number of features for each node
- **S** (*int*) – Number of features for each edge

generate_input_for_resid(*wrapped_pose: menten_gcn.wrappers.WrappedPose*, *resid: int*, *data_cache: Optional[menten_gcn.data_management.DecoratorDataCache] = None*, *sparse: bool = False*, *legal_nbrs: Optional[List[int]] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, List[int]]

Only have 1 focus resid? Then this is sliiiiiiightly cleaner than generate_input(). It's completely debatable if this is even worthwhile

Parameters

- **wrapped_pose** (*WrappedPose*) – Pose to generate data from
- **focus_resid** (*int*) – Which resid is the focus residue? We use Rosetta conventions here, so the first residue is resid #1, second is #2, and so one. No skips.
- **data_cache** (*DecoratorDataCache*) – See make_data_cache for details. It is very important that this cache was created from this pose
- **legal_nbrs** (*list of ints*) – Which resids are allowed to be neighbors? All resids are legal if this is None

Returns

- **X** (*ndarray*) – Node Features
- **A** (*ndarray*) – Adjacency Matrix
- **E** (*ndarray*) – Edge Feature
- **sparse** (*bool*) – This setting will use sparse representations of A and E. X will still have dimension (N,F) but A will now be a scipy.sparse_matrix and E will have dimension (M,S) where M is the number of edges
- **meta** (*list of int*) – Metadata. At the moment this is just a list of resids in the same order as they are listed in X, A, and E

generate_input(*wrapped_pose: menten_gcn.wrappers.WrappedPose*, *focus_resids: List[int]*, *data_cache: Optional[menten_gcn.data_management.DecoratorDataCache] = None*, *sparse: bool = False*, *legal_nbrs: Optional[List[int]] = None*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, List[int]]

This is does the work of creating a graph and representing it as tensors

Parameters

- **wrapped_pose** (*WrappedPose*) – Pose to generate data from
- **focus_resids** (*list of ints*) – Which resids are the focus residues? We use Rosetta conventions here, so the first residue is resid #1, second is #2, and so one. No skips.
- **data_cache** (*DecoratorDataCache*) – See make_data_cache for details. It is very important that this cache was created from this pose
- **sparse** (*bool*) – This setting will use sparse representations of A and E. X will still have dimension (N,F) but A will now be a scipy.sparse_matrix and E will have dimension (M,S) where M is the number of edges
- **legal_nbrs** (*list of ints*) – Which resids are allowed to be neighbors? All resids are legal if this is None

Returns

- **X** (*ndarray*) – Node Features
- **A** (*ndarray*) – Adjacency Matrix
- **E** (*ndarray*) – Edge Feature

- **meta** (*list of int*) – Metadata. At the moment this is just a list of resids in the same order as they are listed in X, A, and E

generate_XAE_input_layers(*sparse: bool = False*) → Tuple[tensorflow.python.keras.engine.base_layer.Layer, tensorflow.python.keras.engine.base_layer.Layer, tensorflow.python.keras.engine.base_layer.Layer]

This is just a safe way to create the input layers for your keras model with confidence that they are the right shape

Parameters **sparse** (*bool*) – If true, returns shapes that work with Spektral's disjoint mode. Otherwise we align with Spektral's batch mode.

Returns

- **X_in** (*Layer*) – Node Feature Input
- **A_in** (*Layer*) – Adjacency Matrix Input
- **E_in** (*Layer*) – Edge Feature Input
- **I_in** (*Layer*) – Batch Index Input (sparse mode only)

make_data_cache(*wrapped_pose:* menten_gcn.wrappers.WrappedPose) → menten_gcn.data_management.DecoratorDataCache

Data caches save time by re-using tensors for nodes and edges you have aleady calculated. This usually gives me a 5-10x speedup but your mileage may vary.

Parameters **wrapped_pose** (*WrappedPose*) – Each pose needs a different cache. Please give us the pose that corresponds to this cache

Returns **cache** (*DecoratorDataCache*) – A data cache that can be passed to generate_input and generate_input_for_resid.

1.3.2 Decorators

class menten_gcn.decorators.**BareBonesDecorator**

This decorator is included in all DataMakers by default. Its goal is to be the starting point upon which everything else is built. It labels focus nodes and labels edges for residues that are polymer bonded to one another.

- 1 Node Feature
- 1 Edge Feature

calc_edge_features(*wrapped_pose*, *resid1*, *resid2*, *dict_cache=None*)

This does all of the business logic of calculating the values to be added for each edge.

This function will never be called in the reverse order (with resid1 and resid2 swapped). Instead, we just create both edges at once.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid1** (*int*) – The first residue ID we are currently generating data for
- **resid1** (*int*) – The second residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns

- **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid1 -> resid2.
- **inv_features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid2 -> resid1.

`calc_node_features`(*wrapped_pose*, *resid*, *dict_cache=None*)

This does all of the business logic of calculating the values to be added for each node.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid** (*int*) – The residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns **features** (*list*) – The length of this list will be the same value as self.n_node_features(). These are the values to represent this decorator's contribution to X for this resid.

`describe_edge_features`()

Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are descriptions of the values to represent this decorator's contribution to E for any arbitrary resid pair.

`describe_node_features`()

Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns **features** (*list*) – The length of this list will be the same value as self.n_node_features(). These are descriptions of the values to represent this decorator's contribution to X for any arbitrary resid.

`get_version_name`()

Get a unique, versioned name of this decorator for maximal reproducability

`n_edge_features`()

How many features will this decorator add to edge tensors (E)?

`n_node_features`()

How many features will this decorator add to node tensors (X)?

Geometry

`class menten_gcn.decorators.CACA_dist`(*use_nm: bool = False*)

Measures distance between the two C-Alpha atoms of each residue

- 0 Node Features
- 1 Edge Feature

Parameters **use_nm** (*bool*) – If true (default), measure distance in Angstroms. Otherwise use nanometers.

`calc_edge_features`(*wrapped_pose*, *resid1: int*, *resid2: int*, *dict_cache=None*)
This does all of the business logic of calculating the values to be added for each edge.

This function will never be called in the reverse order (with resid1 and resid2 swapped). Instead, we just create both edges at once.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid1** (*int*) – The first residue ID we are currently generating data for
- **resid1** (*int*) – The second residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns

- **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid1 -> resid2.
- **inv_features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid2 -> resid1.

`describe_edge_features`()
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns features (*list*) – The length of this list will be the same value as self.n_edge_features(). These are descriptions of the values to represent this decorator's contribution to E for any arbitrary resid pair.

`get_version_name`()
Get a unique, versioned name of this decorator for maximal reproducability

`n_edge_features`()
How many features will this decorator add to edge tensors (E)?

`class` menten_gcn.decorators.`CBCB_dist`(*use_nm: bool = False*)
Measures distance between the two C-Beta atoms of each residue. Note: We will calculate the "ideal ALA" CB location even if this residue has a CB atom. This may sound silly but it is intended to prevents noise from different native amino acid types.

- 0 Node Features
- 1 Edge Feature

Parameters use_nm (*bool*) – If true (default), measure distance in Angstroms. Otherwise use nanometers.

`calc_edge_features`(*wrapped_pose*, *resid1: int*, *resid2: int*, *dict_cache=None*)
This does all of the business logic of calculating the values to be added for each edge.

This function will never be called in the reverse order (with resid1 and resid2 swapped). Instead, we just create both edges at once.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for

- **resid1** (*int*) – The first residue ID we are currently generating data for
- **resid1** (*int*) – The second residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns

- **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid1 -> resid2.
- **inv_features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid2 -> resid1.

`describe_edge_features()`
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns features (*list*) – The length of this list will be the same value as self.n_edge_features(). These are descriptions of the values to represent this decorator's contribution to E for any arbitrary resid pair.

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

`n_edge_features()`
How many features will this decorator add to edge tensors (E)?

`class menten_gcn.decorators.PhiPsiRadians(sincos: bool = False)`
Returns the phi and psi values of each residue position.

- 2-4 Node Features
- 0 Edge Features

Parameters sincos (*bool*) – Return the sine and cosine of phi and psi instead of just the raw values.

`calc_node_features(wrapped_pose, resid, dict_cache=None)`
This does all of the business logic of calculating the values to be added for each node.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid** (*int*) – The residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns features (*list*) – The length of this list will be the same value as self.n_node_features(). These are the values to represent this decorator's contribution to X for this resid.

`describe_node_features()`
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns features (*list*) – The length of this list will be the same value as self.n_node_features(). These are descriptions of the values to represent this decorator's contribution to X for any arbitrary resid.

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

`n_node_features()`
How many features will this decorator add to node tensors (X)?

`class menten_gcn.decorators.ChiAngleDecorator(chi1: bool = True, chi2: bool = True, chi3: bool = True, chi4: bool = True, sincos: bool = True)`
Returns the chi values of each residue position. Ranges from -pi to pi or -1 to 1 if sincos=True.

WARNING: This can behave inconsistantly for proton chis accross modeling frameworks. Rosetta adds hydrogens when they are absent from the input file but MDtraj does not. This results in Rosetta calculating a chi value in some cases that MDtraj skips!

- 0-8 Node Features
- 0 Edge Features

Parameters

- **chi1** (*bool*) – Include chi1's value
- **chi2** (*bool*) – Include chi2's value
- **chi3** (*bool*) – Include chi3's value
- **chi4** (*bool*) – Include chi4's value
- **sincos** (*bool*) – Return the sine and cosine of chi instead of just the raw values

`calc_node_features(wrapped_pose, resid, dict_cache=None)`
This does all of the business logic of calculating the values to be added for each node.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid** (*int*) – The residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns features (*list*) – The length of this list will be the same value as self.n_node_features(). These are the values to represent this decorator's contribution to X for this resid.

`describe_node_features()`
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns features (*list*) – The length of this list will be the same value as self.n_node_features(). These are descriptions of the values to represent this decorator's contribution to X for any arbitrary resid.

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

`n_edge_features()`
How many features will this decorator add to edge tensors (E)?

n_node_features()
How many features will this decorator add to node tensors (X)?

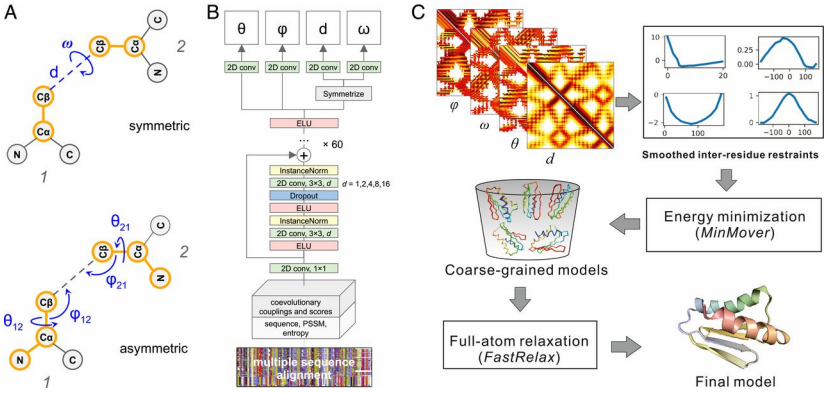
class menten_gcn.decorators.**trRosettaEdges**(*sincos: bool = False*, *use_nm: bool = False*)
Use the residue pair geometries used in this paper: https://www.pnas.org/content/117/3/1496/tab-figures-data

- 0 Node Features
- 4-7 Edge Features

Parameters

- **sincos** (*bool*) – Return the sine and cosine of phi and psi instead of just the raw values.
- **use_nm** (*bool*) – If true, measure distance in Angstroms. Otherwise use nanometers.

  Note: This default value does not match the default of other decorators. This is for the sake of matching the trRosetta paper.



class menten_gcn.decorators.**SimpleBBGeometry**(*use_nm=False*)
Meta-decorator that combines PhiPsiRadians(sincos=False) and CBCB_dist

- 2 Node Features
- 1 Edge Feature

Parameters **use_nm** (*bool*) – If true, measure distance in Angstroms. Otherwise use nanometers.

class menten_gcn.decorators.**StandardBBGeometry**(*use_nm=False*)
Meta-decorator that combines PhiPsiRadians(sincos=True) and trRosettaEdges(sincos=False)

- 4 Node Features
- 4 Edge Features

Parameters **use_nm** (*bool*) – If true, measure distance in Angstroms. Otherwise use nanometers.

class menten_gcn.decorators.**AdvancedBBGeometry**(*use_nm=False*)
Meta-decorator that combines PhiPsiRadians(sincos=True), CACA_dist, and trRosettaEdges(sincos=True)

- 4 Node Features
- 8 Edge Features

Parameters **use_nm** (*bool*) – If true, measure all distances in Angstroms. Otherwise use nanometers.

Sequence

class menten_gcn.decorators.**Sequence**

One-hot encode the canonical amino acid identity on each node.

- 20 Node Features
- 0 Edge Features

calc_node_features(*wrapped_pose*, *resid*, *dict_cache=None*)

This does all of the business logic of calculating the values to be added for each node.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid** (*int*) – The residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns **features** (*list*) – The length of this list will be the same value as self.n_node_features(). These are the values to represent this decorator's contribution to X for this resid.

describe_node_features()

Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns **features** (*list*) – The length of this list will be the same value as self.n_node_features(). These are descriptions of the values to represent this decorator's contribution to X for any arbitrary resid.

get_version_name()

Get a unique, versioned name of this decorator for maximal reproducability

n_edge_features()

How many features will this decorator add to edge tensors (E)?

n_node_features()

How many features will this decorator add to node tensors (X)?

class menten_gcn.decorators.**DesignableSequence**

One-hot encode the canonical amino acid identity on each node, with a 21st value for residues that are not yet assigned an amino acid identity.

Note: requires you to call WrappedPose.set_designable_resids first

- 21 Node Features
- 0 Edge Features

calc_node_features(*wrapped_pose*, *resid*, *dict_cache=None*)

This does all of the business logic of calculating the values to be added for each node.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid** (*int*) – The residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns features (*list*) – The length of this list will be the same value as self.n_node_features(). These are the values to represent this decorator's contribution to X for this resid.

`describe_node_features()`
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns features (*list*) – The length of this list will be the same value as self.n_node_features(). These are descriptions of the values to represent this decorator's contribution to X for any arbitrary resid.

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

`n_edge_features()`
How many features will this decorator add to edge tensors (E)?

`n_node_features()`
How many features will this decorator add to node tensors (X)?

`class menten_gcn.decorators.SequenceSeparation(ln: bool = True)`
The sequence distance between the two residues (i.e., number of residues between these two residues in sequence space, plus one). -1.0 if the two residues belong to different chains.

- 0 Node Features
- 1 Edge Feature

Parameters ln (*bool*) – Report the natural log of the distance instead of the raw count. Does not apply to -1 values

`calc_edge_features(wrapped_pose, resid1, resid2, dict_cache=None)`
This does all of the business logic of calculating the values to be added for each edge.

This function will never be called in the reverse order (with resid1 and resid2 swapped). Instead, we just create both edges at once.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid1** (*int*) – The first residue ID we are currently generating data for
- **resid1** (*int*) – The second residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns

- **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid1 -> resid2.
- **inv_features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid2 -> resid1.

`describe_edge_features()`
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns features (*list*) – The length of this list will be the same value as self.n_edge_features(). These are descriptions of the values to represent this decorator's contribution to E for any arbitrary resid pair.

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

`n_edge_features()`
How many features will this decorator add to edge tensors (E)?

`n_node_features()`
How many features will this decorator add to node tensors (X)?

`class menten_gcn.decorators.SameChain`
1 if the two residues are part of the same protein chain. Otherwise 0.

- 0 Node Features
- 1 Edge Feature

`calc_edge_features(wrapped_pose, resid1, resid2, dict_cache=None)`
This does all of the business logic of calculating the values to be added for each edge.

This function will never be called in the reverse order (with resid1 and resid2 swapped). Instead, we just create both edges at once.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid1** (*int*) – The first residue ID we are currently generating data for
- **resid1** (*int*) – The second residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns

- **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid1 -> resid2.
- **inv_features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid2 -> resid1.

`describe_edge_features()`
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns features (*list*) – The length of this list will be the same value as self.n_edge_features(). These are descriptions of the values to represent this decorator's contribution to E for any arbitrary resid pair.

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

`n_edge_features()`
How many features will this decorator add to edge tensors (E)?

`n_node_features()`
How many features will this decorator add to node tensors (X)?

Rosetta

class menten_gcn.decorators.**RosettaResidueSelectorDecorator**(*selector*, *description: str*)

Takes a user-provided residue selctor and labels each residue with a 1 or 0 accordingly.

- 1 Node Feature
- 0 Edge Features

Parameters

- **selector** (*ResidueSelector*) – This residue selector will be applied to the Rosetta pose
- **description** (*str*) – This is the string that will label this feature in the final summary. Not technically required but highly recommended

Example:

```
import menten_gcn as mg
import menten_gcn.decorators as decs
import pyrosetta

pyrosetta.init()

buried = pyrosetta.rosetta.core.select.residue_selector.LayerSelector()
buried.set_layers( True, False, False )
buried_dec = decs.RosettaResidueSelectorDecorator( selector=buried, description='
↪<Layer select_core="true" />' )

data_maker = mg.DataMaker( decorators=[ buried_dec ], edge_distance_cutoff_A=10.0,␣
↪max_residues=30 )
data_maker.summary()
```

Gives:

```
Summary:

2 Node Features:
1 : 1 if the node is a focus residue, 0 otherwise
2 : 1.0 if the residue is selected by the residue selector, 0.0 otherwise. User␣
↪defined definition of the residue selector and how to reproduce it: <Layer select_
↪core="true" />

1 Edge Features:
1 : 1.0 if the two residues are polymer-bonded, 0.0 otherwise
```

Note that the additional features are due to the BareBonesDecorator, which is included by default

class menten_gcn.decorators.**RosettaResidueSelectorFromXML**(*xml_str: str*, *res_sele_name: str*)

Takes a user-provided residue selctor via XML and labels each residue with a 1 or 0 accordingly.

- 1 Node Feature
- 0 Edge Features

Parameters

- **xml_str** (*str*) – XML snippet that defines the selector

- **res_sele_name** (*str*) – The name of the selector within the snippet

Example:

```
import menten_gcn as mg
import menten_gcn.decorators as decs
import pyrosetta

pyrosetta.init()
xml = '''
<RESIDUE_SELECTORS>
<Layer name="surface" select_surface="true" />
</RESIDUE_SELECTORS>
'''
surface_dec = decs.RosettaResidueSelectorFromXML( xml, "surface" )

max_res=30
data_maker = mg.DataMaker( decorators=[ surface_dec ], edge_distance_cutoff_A=10.0,␣
↪max_residues=max_res )
data_maker.summary()
```

Gives:

```
Summary:

2 Node Features:
1 : 1 if the node is a focus residue, 0 otherwise
2 : 1.0 if the residue is selected by the residue selector, 0.0 otherwise. User␣
↪defined definition of the residue selector and how to reproduce it: Took the␣
↪residue selector named surface from this XML:
<RESIDUE_SELECTORS>
<Layer name="surface" select_surface="true" />
</RESIDUE_SELECTORS>


1 Edge Features:
1 : 1.0 if the two residues are polymer-bonded, 0.0 otherwise
```

Note that the additional features are due to the BareBonesDecorator, which is included by default

class menten_gcn.decorators.**RosettaJumpDecorator**(*use_nm: bool = False*, *rottype: str = 'euler'*)

Measures the translational and rotational relationships between all residue pairs. This uses internal coordinate frames so it is agnostic to the global coordinate system. You can move/rotate your protein around and these will stay the same.

- 0 Node Features
- 6-12 Edge Features

Parameters

- **use_nm** (*bool*) – If true (default), measure distance in Angstroms. Otherwise use nanometers.
- **rottype** (*str*) – How do you want to represent the rotational degrees of freedom? Options are "euler" (default), "euler_sincos", "matrix", "quat", "rotvec", and "rotvec_sincos".

`calc_edge_features`(*wrapped_pose*, *resid1*, *resid2*, *dict_cache=None*)
This does all of the business logic of calculating the values to be added for each edge.

This function will never be called in the reverse order (with resid1 and resid2 swapped). Instead, we just create both edges at once.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid1** (*int*) – The first residue ID we are currently generating data for
- **resid1** (*int*) – The second residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns

- **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid1 -> resid2.
- **inv_features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid2 -> resid1.

`describe_edge_features`()
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are descriptions of the values to represent this decorator's contribution to E for any arbitrary resid pair.

`get_version_name`()
Get a unique, versioned name of this decorator for maximal reproducability

`n_edge_features`()
How many features will this decorator add to edge tensors (E)?

`n_node_features`()
How many features will this decorator add to node tensors (X)?

class menten_gcn.decorators.**RosettaHBondDecorator**(*sfxn=None*, *bb_only: bool = False*)
Takes a user-provided residue selctor via XML and labels each residue with a 1 or 0 accordingly.

- 0 Node Features
- 1-5 Edge Features (depending on bb_only)

Parameters

- **sfxn** (*ScoreFunction*) – Score function used to calculate hbonds. We will use Rosetta's default if this is None
- **bb_only** (*bool*) – Only consider backbone-backbone hbonds. Reduces the number of features from 5 down to 1

class menten_gcn.decorators.**Rosetta_Ref2015_OneBodyEneriges**(*individual: bool = False*, *score_types=None*)
Label each node with its Rosetta one-body energy

- 1 - 20-ish Node Features
- 0 Edge Features

Parameters

- **individual** (*bool*) – If true, list the score for each term individually. Otherwise sum them all into one value.
- **score_types** (*list of ScoreTypes*) – Only use these score types. None (default) includes all default types. Note - this only applies if individual == True

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

`class menten_gcn.decorators.Rosetta_Ref2015_TwoBodyEneriges(individual: bool = False, score_types=None)`
Label each edge with its Rosetta two-body energy

- 0 Node Features
- 1 - 20-ish Edge Features

Parameters

- **individual** (*bool*) – If true, list the score for each term individually. Otherwise sum them all into one value.
- **score_types** (*list of ScoreTypes*) – Only use these score types. None (default) includes all default types. Note - this only applies if individual == True

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

`class menten_gcn.decorators.Ref2015Decorator(individual: bool = False, score_types=None)`
Meta-decorator that combines Rosetta_Ref2015_OneBodyEneriges and Rosetta_Ref2015_TwoBodyEneriges

- 1 - 20-ish Node Features
- 1 - 20-ish Edge Features

Parameters

- **individual** (*bool*) – If true, list the score for each term individually. Otherwise sum them all into one value.
- **score_types** (*list of ScoreTypes*) – Only use these score types. None (default) includes all default types. Note - this only applies if individual == True

`get_version_name()`
Get a unique, versioned name of this decorator for maximal reproducability

1.3.3 Pose Wrappers

Want support for more pose representations? Get in touch!

Note that these are both subclasses of the "WrappedPose" class. So if you see "WrappedPose" in the documentation, that's this!

class menten_gcn.**RosettaPoseWrapper**(*pose*)
This wrapper takes a rosetta pose and requires pyrosetta to be installed

Parameters **pose** (*Pose*) – Rosetta pose

class menten_gcn.**MDTrajPoseWrapper**(*mdtraj_trajectory*)
This wrapper takes a MDTraj trajectory and requires MDTraj to be installed

Parameters **mdtraj_trajectory** (*Trajectory*) – Pose in MDTraj trajectory format

1.4 Examples

1.4.1 Hello World

Let's start simple and just generate a single set of X, A, and E tensors

With PyRosetta

```
import pyrosetta
pyrosetta.init()

import menten_gcn as mg
import menten_gcn.decorators as decs

import numpy as np

# Pick some decorators to add to your network
decorators = [ decs.StandardBBGeometry(), decs.Sequence() ]

data_maker = mg.DataMaker( decorators=decorators,
                           edge_distance_cutoff_A=10.0, # Create edges between all␣
↪residues within 10 Angstroms of each other
                           max_residues=20,             # Do not include more than 20␣
↪residues total in this network
                           nbr_distance_cutoff_A=25.0 ) # Do not include any residue␣
↪that is more than 25 Angstroms from the focus residue(s)

data_maker.summary()

pose = pyrosetta.pose_from_pdb( "test.pdb" )
wrapped_pose = mg.RosettaPoseWrapper( pose )

#picking an arbitrary resid to be interested in
resid_of_interest = 10

X, A, E, resids = data_maker.generate_input_for_resid( wrapped_pose, resid_of_interest )
```

(continues on next page)

(continued from previous page)

```
# Sanity check:
print( "X shape:", X.shape )
print( "A shape:", A.shape )
print( "E shape:", E.shape )
print( "Resids in network:", resids )
```

With MDTraj

```
import mdtraj as md

import menten_gcn as mg
import menten_gcn.decorators as decs

import numpy as np

# Pick some decorators to add to your network
decorators = [ decs.StandardBBGeometry(), decs.Sequence() ]

data_maker = mg.DataMaker( decorators=decorators,
                           edge_distance_cutoff_A=10.0, # Create edges between all␣
↪residues within 10 Angstroms of each other
                           max_residues=20,             # Do not include more than 20␣
↪residues total in this network
                           nbr_distance_cutoff_A=25.0 ) # Do not include any residue␣
↪that is more than 25 Angstroms from the focus residue(s)

data_maker.summary()

pose = md.load_pdb( "test.pdb" )
wrapped_pose = mg.MDTrajPoseWrapper( pose )

#picking an arbitrary resid to be interested in
resid_of_interest = 10

X, A, E, resids = data_maker.generate_input_for_resid( wrapped_pose, resid_of_interest )

# Sanity check:
print( "X shape:", X.shape )
print( "A shape:", A.shape )
print( "E shape:", E.shape )
print( "Resids in network:", resids )
```

1.4.2 Simple Train

This model builds off of the hello world but has some extra complexity and takes us all the way to training

```
import pyrosetta
pyrosetta.init()

import menten_gcn as mg
import menten_gcn.decorators as decs

from spektral.layers import *
from tensorflow.keras.layers import *
from tensorflow.keras.models import Model

import numpy as np

# Pick some decorators to add to your network
decorators = [ decs.StandardBBGeometry(), decs.Sequence() ]

data_maker = mg.DataMaker( decorators=decorators,
                           edge_distance_cutoff_A=10.0, # Create edges between all␣
↪residues within 10 Angstroms of each other
                           max_residues=20,             # Do not include more than 20␣
↪residues total in this network
                           nbr_distance_cutoff_A=25.0 ) # Do not include any residue␣
↪that is more than 25 Angstroms from the focus residue(s)

data_maker.summary()

Xs = []
As = []
Es = []
outs = []

# This part is all very hand-wavy
for pdb in [ "test1.pdb", "test2.pdb", "test3.pdb", "test4.pdb", "test5.pdb" ]:

    pose = pyrosetta.pose_from_pdb( pdb )
    wrapped_pose = mg.RosettaPoseWrapper( pose )
    cache = data_maker.make_data_cache( wrapped_pose )

    for resid in range( 1, pose.size() + 1 ):
        X, A, E, resids = data_maker.generate_input_for_resid( wrapped_pose, resid, data_
↪cache=cache )
        Xs.append( X )
        As.append( A )
        Es.append( E )

        # for the sake of keeping this simple, let's have this model predict if this␣
↪residue is an N-term
        if wrapped_pose.resid_is_N_term( resid ):
            outs.append( [1.0,] )
        else:
```

(continues on next page)

(continued from previous page)

```
            outs.append( [0.0,] )

# Okay now we need to define a model.
# The data_maker can tell use the right sizes to use.
# Better yet, the data_maker can simply create the input layers for us:
X_in, A_in, E_in = data_maker.generate_XAE_input_layers()

# GCN model architectures are tricky
# Here's just a very simple one to get us off the ground

# ECCConv is called EdgeConditionedConv in older versions of spektral
L1 = ECCConv( 30, activation='relu' )([X_in, A_in, E_in])
# Try this if the first one fails:
#L1 = EdgeConditionedConv( 30, activation='relu' )([X_in, A_in, E_in])

L2 = GlobalSumPool()(L1)
L3 = Flatten()(L2)
output = Dense( 1, name="out" )(L3)

model = Model(inputs=[X_in,A_in,E_in], outputs=output)
model.compile(optimizer='adam', loss='binary_crossentropy' )
model.summary()

Xs = np.asarray( Xs )
As = np.asarray( As )
Es = np.asarray( Es )
outs = np.asarray( outs )

print( Xs.shape )
print( As.shape )
print( Es.shape )
print( outs.shape )

model.fit( x=[Xs,As,Es], y=outs, batch_size=32, epochs=10, validation_split=0.2 )
```

1.4.3 Sparse Mode

This modification of the "Simple Train" example utilizes Spektral's disjoint mode to model a sparse representation of the graph.

This can result in lower memory usage depending on the connectivity of your graph.

The key differences are:

- data_maker.generate_graph_for_resid has sparse=True
- data_maker.generate_XAE_input_layers has sparse=True and returns a 4th input
  - inputs=[X_in,A_in,E_in,I_in] when building the model
- We are making a Spektral Dataset and feeding it into the DisjointLoader
- We are using a Spektral Graph instead of freefloating lists. This change can be done with dense mode too.
  - 'y' is the output value in spektral graphs

* Please read Spektral's documentation for options regarding 'y'

```
import pyrosetta
pyrosetta.init()

import menten_gcn as mg
import menten_gcn.decorators as decs

from spektral.layers import *
from tensorflow.keras.layers import *
from tensorflow.keras.models import Model

import numpy as np

decorators = [ decs.StandardBBGeometry(), decs.Sequence() ]

data_maker = mg.DataMaker( decorators=decorators,
                           edge_distance_cutoff_A=10.0, # Create edges between all␣
↪residues within 10 Angstroms of each other
                           max_residues=20,             # Do not include more than 20␣
↪residues total in this network
                           nbr_distance_cutoff_A=25.0 ) # Do not include any residue␣
↪that is more than 25 Angstroms from the focus residue(s)

data_maker.summary()

class MyDataset(spektral.data.dataset.Dataset):
    def __init__(self, **kwargs):
        self.graphs = []
        spektral.data.dataset.Dataset.__init__(self, **kwargs)

    def read(self):
        return self.graphs

dataset = MyDataset()

for pdb in [ "test1.pdb", "test2.pdb", "test3.pdb", "test4.pdb", "test5.pdb" ]:

    pose = pyrosetta.pose_from_pdb( pdb )
    wrapped_pose = mg.RosettaPoseWrapper( pose )
    cache = data_maker.make_data_cache( wrapped_pose )

    for resid in range( 1, pose.size() + 1 ):
        g, resids = data_maker.generate_graph_for_resid( wrapped_pose, resid, data_
↪cache=cache, sparse=True )

        # for the sake of keeping this simple, let's have this model predict if this␣
↪residue is an N-term
        if wrapped_pose.resid_is_N_term( resid ):
             g.y = [1.0,]
        else:
             g.y = [0.0,]
```

(continues on next page)

(continued from previous page)

```
        dataset.graphs.append( g )

# Note we have a 4th input now
X_in, A_in, E_in, I_in = data_maker.generate_XAE_input_layers( sparse=True )

# ECCConv is called EdgeConditionedConv in older versions of spektral
L1 = ECCConv( 30, activation='relu' )([X_in, A_in, E_in])
# Try this if the first one fails:
#L1 = EdgeConditionedConv( 30, activation='relu' )([X_in, A_in, E_in])

L2 = GlobalSumPool()(L1)
L3 = Flatten()(L2)
output = Dense( 1, name="out" )(L3)

# Make sure to include the 4th input because the DisjointLoader will pass it
model = Model(inputs=[X_in,A_in,E_in,I_in], outputs=output)
model.compile(optimizer='adam', loss='binary_crossentropy' )
model.summary()

loader = spektral.data.loaders.DisjointLoader(dataset)
model.fit(loader.load(), steps_per_epoch=loader.steps_per_epoch)
# This part can sometimes fail due to tensorflow / numpy versioning.
# See the troubleshooting page of our documentation for details
```

1.4.4 Custom Decorator

We try and make it relatively easy to create your own decorator. All you need to do is inherit a these seven methods from the base class.

(see full base class description at the bottom)

```
import menten_gcn as mg
import menten_gcn.decorators as decs

 class TestDec( decs.Decorator ):

     def get_version_name( self ):
         return "TestDec"


     # NODES #

     def n_node_features( self ):
         return 1

     def calc_node_features( self, wrapped_protein, resid, dict_cache=None ):
         if wrapped_protein.get_name1( resid ) == "G":
             return [ 1.0 ]
         else:
             return [ 0.0 ]
```

(continues on next page)

(continued from previous page)

```
    def describe_node_features( self ):
        return [ "1 if the residue is GLY, 0 otherwise" ]


    # EDGES #

    def n_edge_features( self ):
        return 2

    def calc_edge_features( self, wrapped_protein, resid1, resid2, dict_cache=None ):
        diff = resid2 - resid1
        same = 1.0 if wrapped_protein.get_name1( resid1 ) == wrapped_protein.get_name1(␣
↪resid2 ) else 0.0
        return [ diff, same ], [ -diff, same ]

    def describe_edge_features( self ):
        return [ "Measures the distance in sequence space between the two residues", "1␣
↪if the two residues have the same amino acid, 0 otherwise" ]



decorators=[ decs.SimpleBBGeometry(), TestDec(), ]
data_maker = mg.DataMaker( decorators=decorators, edge_distance_cutoff_A=10.0, max_
↪residues=5 )
data_maker.summary()
```

```
Summary:

4 Node Features:
1 : 1 if the node is a focus residue, 0 otherwise
2 : Phi of the given residue, measured in radians. Spans from -pi to pi
3 : Psi of the given residue, measured in radians. Spans from -pi to pi
4 : 1 if the residue is GLY, 0 otherwise

4 Edge Features:
1 : 1.0 if the two residues are polymer-bonded, 0.0 otherwise
2 : Euclidean distance between the two CB atoms of each residue, measured in Angstroms.␣
↪In the case of GLY, use an estimate of ALA's CB position
3 : Measures the distance in sequence space between the two residues
4 : 1 if the two residues have the same amino acid, 0 otherwise
```

class menten_gcn.decorators.**Decorator**

n_node_features() → int
How many features will this decorator add to node tensors (X)?

describe_node_features() → List[str]
Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns **features** (*list*) – The length of this list will be the same value as self.n_node_features(). These are descriptions of the values to represent this decorator's contribution to X for any arbitrary resid.

calc_node_features(*wrapped_pose:* menten_gcn.wrappers.WrappedPose, *resid: int*, *dict_cache: Optional[dict] = None*) → List[float]

This does all of the business logic of calculating the values to be added for each node.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid** (*int*) – The residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns **features** (*list*) – The length of this list will be the same value as self.n_node_features(). These are the values to represent this decorator's contribution to X for this resid.

n_edge_features() → int

How many features will this decorator add to edge tensors (E)?

describe_edge_features() → List[str]

Returns descriptions of how each value is computed. Our goal is for these descriptions to be relatively concise but also have enough detail to fully reproduce these calculations.

Returns **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are descriptions of the values to represent this decorator's contribution to E for any arbitrary resid pair.

calc_edge_features(*wrapped_pose:* menten_gcn.wrappers.WrappedPose, *resid1: int*, *resid2: int*, *dict_cache: Optional[dict] = None*) → Tuple[List[float], List[float]]

This does all of the business logic of calculating the values to be added for each edge.

This function will never be called in the reverse order (with resid1 and resid2 swapped). Instead, we just create both edges at once.

Parameters

- **wrapped_pose** (*WrappedPose*) – The pose we are currently generating data for
- **resid1** (*int*) – The first residue ID we are currently generating data for
- **resid1** (*int*) – The second residue ID we are currently generating data for
- **dict_cache** (*dict*) – The same cache that was populated in "cache_data". The user might not have created a cache so don't assume this is not None. See the RosettaHBondDecorator for an example of how to use this

Returns

- **features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid1 -> resid2.
- **inv_features** (*list*) – The length of this list will be the same value as self.n_edge_features(). These are the values to represent this decorator's contribution to E for the edge going from resid2 -> resid1.

get_version_name() → str

Get a unique, versioned name of this decorator for maximal reproducability

cache_data(*wrapped_pose:* menten_gcn.wrappers.WrappedPose, *dict_cache: dict*)

Some decorators can save time by precomputing arbitrary data and storing it in this cache. For example, the RosettaHBondDecorator recomputes and caches all hydrogen bonds so they become a simple lookup when decorating individual nodes and edges.

Parameters

- **wrapped_pose** (*WrappedPose*) – Each pose will be given its own cache. This pose is the one we are currently caching
- **dict_cache** (*dict*) – Destination for your data. Please use a unique key that won't overlap with other decorators'.

For reference, here are the methods for the WrappedPose that will be passed into your decorator

class menten_gcn.**WrappedPose**(*designable_resids=None*)

This is the base class for all pose representations. The internal Menten GCN code will use API listed here

1.5 Technical Overview

docs	
tests	
support	

1.5.1 Documentation

https://menten-gcn.readthedocs.io/

1.5.2 Installation

```
pip install menten-gcn
```

You can also install the in-development version with:

```
pip install https://github.com/MentenAI/menten_gcn/archive/main.zip
```

1.5.3 Development

To run all the tests locally run:

```
tox
```

1.6 Troubleshooting

1.6.1 Sparse Mode

1. Cannot convert a symbolic Tensor to a numpy array

```
NotImplementedError: Cannot convert a symbolic Tensor (gradient_tape/model_1/crystal_
↪conv/sub:0) to a numpy array. This error may indicate that you're trying to pass a␣
↪Tensor to a NumPy call, which is not supported
```

This is a tough one to debug because it is often thrown from deep inside keras. It appears to be solved by updating versions of python, numpy, and/or tensorflow. For example, we might see this pop up in python 3.7 but not 3.6 or 3.8. This stack overflow question also suggests that the numpy/tensorflow relationship could be fixed by downgrading numpy below 1.2.

1.6.2 Versioning

1. No python 3.9 support

We are currently stuck between versions 3.6 and 3.8 of python. 3.5 has reached "end of life" and tensorflow still does not support 3.9. Tensorflow is still listed as a required dependency for Menten GCN but we are working on changing that. Stay tuned!

1.7 Authors

- Menten AI, Inc. - https://menten.ai
  - Created and maintained by Jack Maguire

1.8 Indices and tables

- genindex
- modindex
- search

INDEX

A

AdvancedBBGeometry (*class in menten_gcn.decorators*), 7, 21

B

BareBonesDecorator (*class in menten_gcn.decorators*), 16

C

CACA_dist (*class in menten_gcn.decorators*), 6, 17
cache_data() (*menten_gcn.decorators.Decorator method*), 36
calc_edge_features() (*menten_gcn.decorators.BareBonesDecorator method*), 16
calc_edge_features() (*menten_gcn.decorators.CACA_dist method*), 17
calc_edge_features() (*menten_gcn.decorators.CBCB_dist method*), 18
calc_edge_features() (*menten_gcn.decorators.Decorator method*), 36
calc_edge_features() (*menten_gcn.decorators.RosettaJumpDecorator method*), 26
calc_edge_features() (*menten_gcn.decorators.SameChain method*), 24
calc_edge_features() (*menten_gcn.decorators.SequenceSeparation method*), 23
calc_node_features() (*menten_gcn.decorators.BareBonesDecorator method*), 17
calc_node_features() (*menten_gcn.decorators.ChiAngleDecorator method*), 20
calc_node_features() (*menten_gcn.decorators.Decorator method*), 35
calc_node_features() (*menten_gcn.decorators.DesignableSequence method*), 22
calc_node_features() (*menten_gcn.decorators.PhiPsiRadians method*), 19
calc_node_features() (*menten_gcn.decorators.Sequence method*), 22
CBCB_dist (*class in menten_gcn.decorators*), 6, 18
ChiAngleDecorator (*class in menten_gcn.decorators*), 6, 20

D

DataMaker (*class in menten_gcn*), 13
Decorator (*class in menten_gcn.decorators*), 35
describe_edge_features() (*menten_gcn.decorators.BareBonesDecorator method*), 17
describe_edge_features() (*menten_gcn.decorators.CACA_dist method*), 18
describe_edge_features() (*menten_gcn.decorators.CBCB_dist method*), 19
describe_edge_features() (*menten_gcn.decorators.Decorator method*), 36
describe_edge_features() (*menten_gcn.decorators.RosettaJumpDecorator method*), 27
describe_edge_features() (*menten_gcn.decorators.SameChain method*), 24
describe_edge_features() (*menten_gcn.decorators.SequenceSeparation method*), 23
describe_node_features() (*menten_gcn.decorators.BareBonesDecorator method*), 17
describe_node_features() (*menten_gcn.decorators.ChiAngleDecorator method*), 20

describe_node_features() (*menten_gcn.decorators.Decorator method*), 35
describe_node_features() (*menten_gcn.decorators.DesignableSequence method*), 23
describe_node_features() (*menten_gcn.decorators.PhiPsiRadians method*), 19
describe_node_features() (*menten_gcn.decorators.Sequence method*), 22
DesignableSequence (*class in menten_gcn.decorators*), 8, 22

G

generate_input() (*menten_gcn.DataMaker method*), 15
generate_input_for_resid() (*menten_gcn.DataMaker method*), 14
generate_XAE_input_layers() (*menten_gcn.DataMaker method*), 16
get_N_F_S() (*menten_gcn.DataMaker method*), 14
get_version_name() (*menten_gcn.decorators.BareBonesDecorator method*), 17
get_version_name() (*menten_gcn.decorators.CACA_dist method*), 18
get_version_name() (*menten_gcn.decorators.CBCB_dist method*), 19
get_version_name() (*menten_gcn.decorators.ChiAngleDecorator method*), 20
get_version_name() (*menten_gcn.decorators.Decorator method*), 36
get_version_name() (*menten_gcn.decorators.DesignableSequence method*), 23
get_version_name() (*menten_gcn.decorators.PhiPsiRadians method*), 20
get_version_name() (*menten_gcn.decorators.Ref2015Decorator method*), 28
get_version_name() (*menten_gcn.decorators.Rosetta_Ref2015_OneBodyEneriges method*), 28
get_version_name() (*menten_gcn.decorators.Rosetta_Ref2015_TwoBodyEneriges method*), 28
get_version_name() (*menten_gcn.decorators.RosettaJumpDecorator method*), 27
get_version_name() (*menten_gcn.decorators.SameChain method*), 24
get_version_name() (*menten_gcn.decorators.Sequence method*), 22
get_version_name() (*menten_gcn.decorators.SequenceSeparation method*), 24

M

make_data_cache() (*menten_gcn.DataMaker method*), 16
MDTrajPoseWrapper (*class in menten_gcn*), 29

N

n_edge_features() (*menten_gcn.decorators.BareBonesDecorator method*), 17
n_edge_features() (*menten_gcn.decorators.CACA_dist method*), 18
n_edge_features() (*menten_gcn.decorators.CBCB_dist method*), 19
n_edge_features() (*menten_gcn.decorators.ChiAngleDecorator method*), 20
n_edge_features() (*menten_gcn.decorators.Decorator method*), 36
n_edge_features() (*menten_gcn.decorators.DesignableSequence method*), 23
n_edge_features() (*menten_gcn.decorators.RosettaJumpDecorator method*), 27
n_edge_features() (*menten_gcn.decorators.SameChain method*), 24
n_edge_features() (*menten_gcn.decorators.Sequence method*), 22
n_edge_features() (*menten_gcn.decorators.SequenceSeparation method*), 24
n_node_features() (*menten_gcn.decorators.BareBonesDecorator method*), 17
n_node_features() (*menten_gcn.decorators.ChiAngleDecorator method*), 20
n_node_features() (*menten_gcn.decorators.Decorator method*), 35
n_node_features() (*menten_gcn.decorators.DesignableSequence method*), 23
n_node_features() (*menten_gcn.decorators.PhiPsiRadians method*), 20
n_node_features() (*menten_gcn.decorators.RosettaJumpDecorator method*), 27
n_node_features() (*menten_gcn.decorators.SameChain method*), 24
n_node_features() (*menten_gcn.decorators.Sequence method*), 22
n_node_features() (*menten_gcn.decorators.SequenceSeparation method*), 24

P

PhiPsiRadians (*class in menten_gcn.decorators*), 6, 19

R

Ref2015Decorator (*class in menten_gcn.decorators*), 11, 28
Rosetta_Ref2015_OneBodyEneriges (*class in menten_gcn.decorators*), 10, 27
Rosetta_Ref2015_TwoBodyEneriges (*class in menten_gcn.decorators*), 11, 28
RosettaHBondDecorator (*class in menten_gcn.decorators*), 10, 27

RosettaJumpDecorator (*class in menten_gcn.decorators*), 10, 26
RosettaPoseWrapper (*class in menten_gcn*), 29
RosettaResidueSelectorDecorator (*class in menten_gcn.decorators*), 8, 25
RosettaResidueSelectorFromXML (*class in menten_gcn.decorators*), 9, 25

S

SameChain (*class in menten_gcn.decorators*), 8, 24
Sequence (*class in menten_gcn.decorators*), 8, 22
SequenceSeparation (*class in menten_gcn.decorators*), 8, 23
SimpleBBGeometry (*class in menten_gcn.decorators*), 7, 21
StandardBBGeometry (*class in menten_gcn.decorators*), 7, 21
summary() (*menten_gcn.DataMaker method*), 13

T

trRosettaEdges (*class in menten_gcn.decorators*), 6, 21

W

WrappedPose (*class in menten_gcn*), 37