PlasmoData

David Cole

March 7, 2024

Contents

Part I

Introduction

Chapter 1

PlasmoData

PlasmoData.jl is a package for Julia designed for representing and modeling data as graphs and for building graph models that contain large amounts of data on the nodes or edges of the graph. This package also has an accompanying package DataGraphPlots.jl which can be used for plotting the graphs.

1.1 Installation

To install this package, you can use

```
using Pkg
Pkg.add(url="https://github.com/zavalab/PlasmoData.jl")
```

or

```
pkg> add https://github.com/zavalab/PlasmoData.jl
```

1.2 Overview

PlasmoData.jl is designed to store data within the graph structure and to manipulate that graph based on the data. It extends the package Graphs.jl, which is a highly optimized and efficient package in Julia. PlasmoData.jl enables representing datasets (such as matrices, images, or tensors) as graphs and for performing some topological data analysis (TDA). Some of these concepts can be found in this paper.

PlasmoData.jl uses an object `DataGraph` (or `DataDiGraph` for directed graphs) to store information. These objects contain the following features:

- `g`: `SimpleGraph` (or `SimpleDiGraph` for directed graphs) containing the graph structure.
- `nodes`: A vector of nodes, where the entries of the vector are node names. These names are of type `Any` so that the nodes can use a variety of naming conventions (strings, symbols, tuples, etc.)
- `edges`: A vector of tuples, where each tuple contains two entries, where each entry relates to a node.
- `node_map`: A dictionary that maps the node names to their index in the `nodes` vector
- `edge_map`: A dictionary that maps the edges to their index in the `edges` vector.

- `node_data`: An object of type `NodeData` that includes a matrix of data, where the first dimension of the matrix corresponds to the node, and the second dimension corresponds to attributes for the nodes. Any number of attributes is allowed, and `NodeData` also includes attribute names and a mapping of the attribute name to the column of the data matrix.
- `edge_data`: An object of type `EdgeData` that includes a matrix of data, where the first dimension fo the matrix corresponds to the edges, and the second dimension corresponds to attributes for the edges. Any number of attributes is allowed, and `EdgeData` also includes attribute names and a mapping of the attribute name to the column of the data matrix.
- `graph_data`: An object of type `GraphData` that includes a vector of data whose dimension corresponds to the number of attributes for the graph. Any number of attributes is allowed, and `GraphData` also includes attribute names and a mapping of the attribute name to the entry in the vector.

PlasmoData.jl includes several functions for building graphs from specific data structures, including functions like `matrix_to_graph`, `symmetric_matrix_to_graph`, and `tensor_graph` which build specific graph structures an save data to those structures.

PlasmoData.jl also includes functions for manipulating graph structure and analyzing the resulting topology of those structures. Functions `filter_nodes`, `filter_edges`, or `aggregate` change the graph structure based on the arguments passed to the functions. There are also functions such as `get_EC`, `run_EC_on_nodes`, and `run_EC_on_edges` that get the Euler Characteristic or the Euler Characteristic Curve for a graph, and other functions such as `cycle_basis`, `diameter`, or `average_degree` (largely extensions of Graphs.jl) for finding other topological descriptors.

Support for `DataDiGraphs` is still underway. However, for `DataGraph` objects, all functions shown above have doc strings, which can be accessed through the REPL by first typing `?` and then the function or object name.

1.3 Bug Reports and Support

This package is under development, and significant changes will continue to come. If you encounter any issues or bugs, please submit them through the Github issue tracker.

Part II

Quick Start

Chapter 2

Getting Started

A DataGraph can be initiated by calling

```
dg = DataGraph()
```

PlasmoData.jl also supports building a DataGraph from an adjacency matrix. The DataGraph can be changed by adding nodes or edges to the graph, as shown below. add_node! takes two arguments: the DataGraph of interest and the node name (any data type is permitted). add_edge takes three arguments, the DataGraph of interest, and the names of two nodes in the graph.

```
add_node!(dg, "node1")
add_node!(dg, :node2)
add_node!(dg, 3)

add_edge!(dg, "node1", :node2)
add_edge!(dg, 3, :node2)
add_edge!(dg, "node1", 3)
```

Data can be added to these nodes or edges by calling add_node_data! or add_edge_data! as shown below. Here, these functions take similar arguments to add_node! or add_edge!, but they also take two additional arguments, one for the weight value and one for the attribute name (must be a string). When setting a new attribute, the other nodes or edges will receive a default value of 0.

```
add_node_data!(dg, "node1", 1.0,   "node_weight_1")
add_node_data!(dg, :node2,  2.0, "node_weight_1")
add_node_data!(dg, 3,       3.0,   "node_weight_1")

add_edge_data!(dg, "node1", :node2,  4.0, "edge_weight_1")
add_edge_data!(dg, :node2,  3,       5.0, "edge_weight_1")
add_edge_data!(dg, 3,       "node1", 6.0, "edge_weight_1")
```

Note that for DataGraphs, the order of the nodes in the edge is not important, but it is important for DataDiGraphs.

2.1 Additional Functions for Building Graphs

There are also functions for direclty building a graph from a set of data. Examples are shown below.

```
random_matrix = rand(20, 20)

matrix_graph = matrix_to_graph(random_matrix, "matrix_weight")

symmetric_random_matrix = random_matrix .+ random_matrix'

symmetric_matrix_graph = symmetric_matrix_to_graph(symmetric_random_matrix, "matrix_weight")

random_tensor = rand(20, 20, 15)

tensor_graph = tensor_to_graph(random_tensor)

matrix_graph_multiple_weights = matrix_to_graph(random_tensor)
```

2.2 Manipulating Graph Structure

PlasmoData.jl enables manipulating graph structure while maintaining the data in the resulting graph. For both `DataGraphs` and `DataDiGraphs`, users can call `filter_nodes`, `filter_edges`, `aggregate`, `remove_node!`, and `remove_edge!`.

```
# Keep nodes for which the "matrix_weight" is greater than 0.5
filtered_graph = filter_nodes(matrix_graph, 0.5, "matrix_weight", fn = Base.isgreater)

# Keep edges for which the "matrix_weight" is less than 0.5
filtered_graph = filter_edges(symmetric_matrix_graph, 0.5, "matrix_weight", fn = Base.isless)

# aggregate the nodes (2, 3), (2, 4), and (3, 3) together
aggregated_graph = aggregate(matrix_graph, [(2, 3), (2, 4), (3, 3)], "new_node")

remove_node!(matrix_graph, (4, 4))
remove_edge!(matrix_graph, (4, 5), (4, 6))
```

2.3 Data Analysis

Representing and modeling data as a graph enables unique analysis, including analyzing the topology of the resulting graph. Topological Data Analysis (TDA) can be applied generally to geometric shapes, including to graphs. TDA is an expanding field, and it has been shown to be a powerful data analysis tool for many systems. Some TDA is enabled within PlasmoData.jl (in many cases, through extending the functions of Graphs.jl).

The Euler Characteristic (EC) is a topological descriptor for geometric objects. For graphs, the EC is equal to the number of nodes minus the number of edges (or equivalently, the number of connected components minus the number of cycles). Often, the EC is combined with filtration to form an EC Curve. For node- or edge-weighted graphs, this involves filtering out nodes or edges of a graph based on their weight value and computing the EC of the resulting structure. This is done at a range of threshold values to get a vector (curve). PlasmoData.jl provides functions for computing the EC and the EC curve (note that the EC only applies to `DataGraphs` and not `DataDiGraphs`)

```
thresh = 0:.01:1

EC_curve = run_EC_on_nodes(matrix_graph, thresh)
```

```
EC_curve = run_EC_on_edges(symmetric_matrix_graph, thresh)

EC = get_EC(matrix_graph)
```

Other metrics are also available for studying the topology of the graph. Some examples are shown below (largely extensions of Graphs.jl)

```
ad = average_degree(matrix_graph)

cycles = Graphs.cycle_basis(matrix_graph)

conn_comp = Graphs.connected_components(matrix_graph)

neighbor_list = Graphs.neighbors(matrix_graph, (2, 2))

diam = Graphs.diameter(matrix_graph)

communities = Graphs.clique_percolation(matrix_graph, k = 3)

max_clique = Graphs.maximal_cliques(matrix_graph)
```

In addition, there are also functions for analyzing the connections in directed graphs. These include has_path (returns `true` or `false` depending on if there is a path between two given nodes) and get_path (returns the path between two given nodes if it exists). In addition, these functions can also take another argument of an intermediate node (i.e., for detecting a path between two nodes that passes through the intermediate node). In addition, there are functions to get all the upstream and downstream nodes of a given node using upstream_nodes and downstream_nodes.

```
nodes = [1, 2, 3, 4, 5]
edges = [(1, 2), (1, 3), (2, 3), (3, 4), (4, 2), (5, 4)]

dg = DataDiGraph()
for i in nodes
    add_node!(dg, i)
end

for i in edges
    add_edge!(dg, i)
end

# Test if there is a path between nodes 1 and 4
PlasmoData.has_path(dg, 1, 4)

# Test if there is a path between nodes 1 and 5 that passes through 4
PlasmoData.has_path(dg, 1, 4, 5)

# Return the (shortest) path between Nodes 1 and 4
get_path(dg, 1, 4)

# Return the (shortest) path between Nodes 1 and 4 that passes through 2
get_path(dg, 1, 2, 4)

up_nodes = upstream_nodes(dg, 4)
```

```
down_nodes = downstream_nodes(dg, 1
```

2.4 Further Examples

To see additional examples of how PlasmoData.jl can be used, please see the examples directory within this repository.

Part III

API Manual

Chapter 3

API Manual

`PlasmoData.DataGraphUnion` – Type.

```
DataGraphUnion
```

Data type that is a union of DataGraph and DataDiGraph; used for functions that apply to both data types

source

`PlasmoData.DataDiGraph` – Type.

```
DataDiGraph{T, T1, T2, T3, M1, M2}
```

Object for building and storing directed graphs that contain numerical data on nodes and/or edges.

DataDiGraphs have the following attributes: `g`: Graphs.SimpleDiGraph Object `nodes`: Vector of node names; node names are of type Any `edges`: Vector of edges; edges are tuples of integers `node_map`: dictionary pointing node name to node number `edge_map`: dictionary pointing tuple (nodename1, nodename2) to (nodenumber1, nodenumber2) `node_data`: NodeData object with attributes and data `edge_data`: EdgeData object with attributes and data `graph_data`: GraphData object with attributes and data

source

`PlasmoData.DataDiGraph` – Method.

```
DataDiGraph(adjacency_matrix::AbstractMatrix)
```

Constructor for building a DataDiGraph object from an adjacency matrix.

source

`PlasmoData.DataDiGraph` – Method.

```
DataDiGraph{T, T1, T2, M1, M2}()
DataDiGraph()
```

Constructor for initializing and empty DataDiGraph object. Datatypes are as follows: T is the integer type for indexing, T1 and T2 are the data type in the node and edge data respectively, and M1 <: AbstractMatrix{T1} corresponds to the node data and M2 <: AbstractMatrix{T2} corresponds to the edge data.

When T, T1, T2, M1, and M2 are not defined, the defaults are `Int`, `Float64`, `Float64`, `Matrix{Float64}`, and `Matrix{Float64}` respectively.

source

`PlasmoData.DataGraph` – Type.

```
DataGraph{T, T1, T2, T3, M1, M2}
```

Object for building and storing undirected graphs that contain numerical data on nodes and/or edges.

DataGraphs have the following attributes: `g`: Graphs.SimpleGraph Object `nodes`: Vector of node names; node names are of type `Any` `edges`: Vector of edges; edges are tuples of integers `node_map`: dictionary pointing node name to node number `edge_map`: dictionary pointing tuple (nodename1, nodename2) to (nodenumber1, nodenumber2) `node_data`: NodeData object with attributes and data `edge_data`: EdgeData object with attributes and data `graph_data`: GraphData object with attributes and data

source

`PlasmoData.DataGraph` – Method.

```
DataGraph(adjacency_matrix::AbstractMatrix)
```

Constructor for building a DataGraph object from an adjacency matrix.

source

`PlasmoData.DataGraph` – Method.

```
DataGraph(edge_list)
```

Constructor for building a DataGraph object from a list of edges, where the edge list is a vector of Tuple{Any, Any}.

source

`PlasmoData.DataGraph` – Method.

```
DataGraph(nodes, edges; kwargs)
```

Constructor for building a DataGraph object from a list of nodes and edges. Key word arguments include `ne`, `fadjlist`, `node_attributes`, `edge_attributes`, `node_map`, `edge_map`, `node_data`, and `edge_data`.

source

`PlasmoData.DataGraph` – Method.

```
DataGraph{T, T1, T2, T3, M1, M2}()
DataGraph()
```

Constructor for initializing and empty DataGraph object. Datatypes are as follows: T is the integer type for indexing, T1, T2, and T3 are the data type in the node, edge, and graph data respectively, and M1 <: AbstractMatrix{T1} corresponds to the node data and M2 <: AbstractMatrix{T2} corresponds to the edge data.

When T, T1, T2, T3, M1, and M2 are not defined, the defaults are `Int`, `Float64`, `Float64`, `Float64`, `Matrix{Float64}`, and `Matrix{Float64}` respectively.

source

`PlasmoData.EdgeData` – Type.

```
EdgeData{T, T2, M2}
```

Object for building and storing data corresponding to the edges of a graph. Data is stored in a matrix, but columns of the matrix have attribute names stored in this struct

EdgeData have the following attributes: `attributes`: vector of strings with length equal to the number of columns of `data`. Each entry is the name of the attribute of that column of data `attribute_map`: dictionary with keys matching the entries of `attributes`. Maps the key to the corresponding column index `data`: Matrix with the number of rows corresponding to the number of edgess in the graph and with a column for each attribute in `attributes`

source

`PlasmoData.EdgeData` – Method.

```
EdgeData(attributes = Vector{String}(),
    attribute_map = Dict{String, Int}(),
    data = Array{Float64}(undef, (0, 0))
)
```

Constructor for building EdgeData{T, T2, M2}

source

`PlasmoData.GraphData` – Type.

```
GraphData{T, T2, M2}
```

Object for building and storing data corresponding to a graph. Data is stored in a vector, but entries of the vector have attribute names stored in this struct

GraphData have the following attributes: `attributes`: vector of strings with length equal to the length of `data`. Each entry is the name of the attribute of that column of data `attribute_map`: dictionary with keys matching the entries of `attributes`. Maps the key to the corresponding vector index `data`: Vector with length equal to the number of attributes

source

`PlasmoData.GraphData` – Method.

```
GraphData(attributes = Vector{String}(),
    attribute_map = Dict{String, Int}(),
    data = Vector{Float64}(undef, 0)
)
```

Constructor for building GraphData{T, T3}

source

`PlasmoData.NodeData` – Type.

```
NodeData{T, T1, M1}
```

Object for building and storing data corresponding to the nodes of a graph. Data is stored in a matrix, but columns of the matrix have attribute names stored in this struct

NodeData have the following attributes: `attributes`: vector of strings with length equal to the number of columns of `data`. Each entry is the name of the attribute of that column of data `attribute_map`: dictionary with keys matching the entries of `attributes`. Maps the key to the corresponding column index `data`: Matrix with the number of rows corresponding to the number of nodes in the graph and with a column for each attribute in `attributes`

source

`PlasmoData.NodeData` – Method.

```
NodeData(attributes = Vector{String}(),
    attribute_map = Dict{String, Int}(),
    data = Array{Float64}(undef, (0, 0))
)
```

Constructor for building NodeData{T, T1, M1}

source

`PlasmoData._build_matrix_graph!` – Method.

```
_build_matrix_graph!
```

Constructs the graph structure for the function `matrix_to_graph`

source

`PlasmoData._get_bfs_dfs_list` – Method.

```
_get_bfs_dfs_list(dg::DataDiGraph, node, algorithm, dir)
```

Returns the list of nodes that are upstream or downstream of node in the DataDiGraph dg

source

PlasmoData.add_edge! – Method.

```
add_edge!(dg, node_1, node_2)
add_edge!(dg, (node1, node2))
```

Add an edge to the DataDiGraph, dg. If the nodes are not defined in the graph, they are added to the graph

source

PlasmoData.add_edge! – Method.

```
add_edge!(dg, node_1, node_2)
add_edge!(dg, (node1, node2))
```

Add an edge to the DataGraph, dg. If the nodes are not defined in the graph, they are added to the graph

source

PlasmoData.add_edge_attribute! – Method.

```
add_edge_attribute!(datagraph, attribute, default_weight = 0.0)
add_edge_attribute!(datadigraph, attribute, default_weight = 0.0)
```

Add a column filled with `default_weight` to the edge_data matrix with the name `attribute`. If `attribute` already exists in the edge data, an error is thrown.

source

PlasmoData.add_edge_data! – Method.

```
add_edge_data!(datadigraph, node_name1, node_name2, edge_weight, attribute_name)
add_edge_data!(datadigraph, edge, edge_weight, attribute_name)
```

Add a weight value for the edge between nodename1 and nodename2 in the DataDiGraph object. When using the second function, edge must be a tuple with two node names. User must pass an "attribute name" for the given weight. All other edges that do not have an edge_weight value defined for that attribute name default to a value of zero.

source

PlasmoData.add_edge_data! – Method.

```
add_edge_data!(datagraph, node_name1, node_name2, edge_weight, attribute_name)
add_edge_data!(datagraph, edge, edge_weight, attribute_name)
```

Add a weight value for the edge between nodename1 and nodename2 in the DataGraph object. When using the second function, edge must be a tuple with two node names. User must pass an "attribute name" for the given weight. All other edges that do not have an edge_weight value defined for that attribute name default to a value of zero.

source

`PlasmoData.add_edge_dataset!` – Method.

```
add_edge_dataset!(dg::D, weight_dict, attribute) where {D <: DataGraphUnion}
```

Add the data in `weight_dict` as edge data on `dg` under the name `attribute`. `weight_dict` must contain keys that correspond to the edges (as node names, not integers) in `dg.edges`.

source

`PlasmoData.add_edge_dataset!` – Method.

```
add_edge_dataset!(dg::D, weight_list, attribute) where {D <: DataGraphUnion}
```

Add the entries of `weight_list` as edge data on `dg` under the name `attribute`. `weight_list` must be the same length as the number of edges in `dg`. Entries of `weight_list` will be added as edge data in the order that edges are listed in `dg.edges`.

source

`PlasmoData.add_edge_dataset!` – Method.

```
add_edge_dataset!(dg::D, edge_list, weight_list, attribute) where {D <: DataGraphUnion}
```

Add the edge data in `weight_list` to `dg`. `edge_list` is a list of edges (as node names, not integers) in `dg` and `weight_list` is a list of data/objects to be saved as edge data under the name `attribute`. `edge_list` and `weight_list` must have the same length, and entries of `weight_list` will be added to the corresponding edge in `edge_list`

source

`PlasmoData.add_graph_data!` – Method.

```
add_graph_data!(dg::D, weight, attribute) where {D <: DataGraphUnion}
```

Add the value `weight` to the graph under the name attribute. If the attribute is already defined, the value will be reset to `weight`.

source

`PlasmoData.add_node!` – Method.

```
add_node!(dg, node_name)
```

Add the node node_name to the DataDiGraph dg

source

PlasmoData.add_node! – Method.

```
add_node!(dg, node_name)
```

Add the node node_name to the DataGraph dg

source

PlasmoData.add_node_attribute! – Method.

```
add_node_attribute!(datagraph, attribute, default_weight = 0.0)
add_node_attribute!(datadigraph, attribute, default_weight = 0.0)
```

Add a column filled with default_weight to the node_data matrix with the name attribute. If attribute already exists in the node data, an error is thrown.

source

PlasmoData.add_node_data! – Method.

```
add_node_data!(dg::D, node_name, node_weight, attribute_name) where {D <: DataGraphUnion}
```

Add a weight value for the given node name in the DataGraph object. User must pass an "attribute name" for the given weight. All other nodes that do not have a node_weight value defined for that attribute name default to a value of zero.

source

PlasmoData.add_node_dataset! – Method.

```
add_node_dataset!(dg::D, weight_dict, attribute) where {D <: DataGraphUnion}
```

Add the data in weight_dict as node data on dg under the name attribute. weight_dict must contain keys that correspond to the node names in dg.nodes.

source

PlasmoData.add_node_dataset! – Method.

```
add_node_dataset!(dg::D, weight_list, attribute) where {D <: DataGraphUnion}
```

Add the entries of `weight_list` as node data on `dg` under the name `attribute`. `weight_list` must be the same length as the number of nodes in `dg`. Entries of `weight_list` will be added as node data in the order that nodes are listed in `dg.nodes`.

source

`PlasmoData.add_node_dataset!` – Method.

```
add_node_dataset!(dg::D, node_list, weight_list, attribute) where {D <: DataGraphUnion}
```

Add the node data in `weight_list` to `dg`. `node_list` is a list of nodes in `dg` and `weight_list` is a list of values/things to be saved as node data under the name `attribute`. `node_list` and `weight_list` must have the same length, and entries of `weight_list` will be added to the corresponding node in `node_list`

source

`PlasmoData.adjacency_matrix` – Method.

```
adjacency_matrix(datagraph)
adjacency_matrix(datadigraph)
```

Return the adjacency matrix of a DataGraph object

source

`PlasmoData.aggregate` – Method.

```
aggregate(datadigraph, node_list, aggregated_node_name;
            node_fn = mean, edge_fn = mean, save_agg_edge_data = false,
            agg_edge_fn = mean, agg_edge_val = 0, node_attributes_to_add = String[]
)
```

Aggregates all the nodes in `node_list` into a single node which is called `aggregated_node_name`. If nodes have any weight/attribute values defined, these values are combined via the `node_fn` function. The default for `node_fn` is Statistics.mean which averages the data for the nodes in `node_list`. Edge data are also are also combined via the `edge_fn` when two or more nodes in the `node_list` are connected to the same node and these edges have data defined on them. The `edge_fn` also defaults to `Statistics.mean`

If edges exist between nodes in `node_list`, the data on these edges can optionally be saved on the `aggregated_node_name` node by setting `save_agg_edge_data = true`. If true, then the edge data on these edges is aggregated using `agg_edge_fn`. If the user wants to define new attribute names for this data, they can pass a vector to `node_attributes_to_add`; if no vector is defined, the data will be aggregated under the names of the `edge_data` attributes. All other nodes except the aggregated nodes will have these attributes initialized as `agg_edge_val`.

source

`PlasmoData.aggregate` – Method.

```
aggregate(datagraph, node_list, aggregated_node_name;
            node_fn = mean, edge_fn = mean, save_agg_edge_data = false,
            agg_edge_fn = mean, agg_edge_val = 0, node_attributes_to_add = String[]
)
```

Aggregates all the nodes in `node_list` into a single node which is called `aggregated_node_name`. If nodes have any weight/attribute values defined, these values are combined via the `node_fn` function. The default for `node_fn` is Statistics.mean which averages the data for the nodes in `node_list`. Edge data are also are also combined via the `edge_fn` when two or more nodes in the `node_list` are connected to the same node and these edges have data defined on them. The `edge_fn` also defaults to `Statistics.mean`

If edges exist between nodes in `node_list`, the data on these edges can optionally be saved on the `aggregated_node_name` node by setting `save_agg_edge_data = true`. If true, then the edge data on these edges is aggregated using `agg_edge_fn`. If the user wants to define new attribute names for this data, they can pass a vector to `node_attributes_to_add`; if no vector is defined, the data will be aggregated under the names of the `edge_data` attributes. All other nodes except the aggregated nodes will have these attributes initialized as `agg_edge_val`.

source

`PlasmoData.average_degree` – Method.

```
average_degree(datagraph)
```

Returns the average degree for `datagraph`

source

`PlasmoData.downstream_nodes` – Method.

```
downstream_nodes(dg::DataDiGraph, node; algorithm = "bfs")
```

Return a list of all the nodes that are downstream of `node` in the DataDiGraph `dg`. Algorithm options are "bfs" and "dfs"

source

`PlasmoData.filter_edges` – Method.

```
filter_edges(datadigraph, filter_value, attribute = dg.edge-data.attributes[1]; fn = isless)
```

Removes the edges of the graph whose weight value of `attribute` is greater than the given `filter_value`. If `attribute` is not specified, this defaults to the first attribute within the DataGraph's EdgeData.

`fn` is a function that takes an input of two scalar values and is broadcast to the data vector. For example, isless, isgreater, isequal

source

`PlasmoData.filter_edges` – Method.

```
filter_edges(datagraph, filter_value, attribute = dg.edge_data.attributes[1]; fn = isless)
```

Removes the edges of the graph whose weight value of `attribute` is greater than the given `filter_value`. If `attribute` is not specified, this defaults to the first attribute within the DataGraph's EdgeData.

`fn` is a function that takes an input of two scalar values and is broadcast to the data vector. For example, isless, isgreater, isequal

source

`PlasmoData.filter_nodes` – Method.

```
filter_nodes(datadigraph, filter_value, attribute = dg.node_data.attributes[1]; fn = isless)
```

Removes the nodes of the graph whose weight value of `attribute` is greater than the given `filter_value`. If `attribute` is not specified, this defaults to the first attribute within the DataGraph's NodeData.

`fn` is a function that takes an input of two scalar values and is broadcast to the data vector. For example, isless, isgreater, isequal

source

`PlasmoData.filter_nodes` – Method.

```
filter_nodes(datagraph, filter_value, attribute = dg.node_data.attributes[1]; fn = isless)
```

Removes the nodes of the graph whose weight value of `attribute` is greater than the given `filter_value`. If `attribute` is not specified, this defaults to the first attribute within the DataGraph's NodeData.

`fn` is a function that takes an input of two scalar values and is broadcast to the data vector. For example, isless, isgreater, isequal

source

`PlasmoData.get_EC` – Method.

```
get_EC(datagraph)
```

Returns the Euler Characteristic for a DataGraph. The Euler Characteristic is equal to the number of nodes minus the number of edges or the number of connected components minus the number of cycles

source

`PlasmoData.get_edge_attributes` – Method.

```
get_edge_attributes(dg::D) where {D <: DataGraphUnion}
```

Returns the list of attributes contained in the EdgeData of `dg`

source

PlasmoData.get_edge_data – Method.

```
get_edge_data(dg::D) where {D <: DataGraphUnion}
```

Returns the data object from a DataGraph's or DataDiGraph's EdgeData

source

PlasmoData.get_edge_data – Method.

```
get_edge_data(dg::D, attribute) where {D <: DataGraphUnion}
```

Returns a vector of the edge data corresponding to the attribute

source

PlasmoData.get_edge_data – Method.

```
get_edge_data(dg::D, attribute_list; edges) where {D <: DataGraphUnion}
```

Returns a matrix of the edge data for the attributes in attribute list in the order that the attributes are defined in the list

source

PlasmoData.get_edge_data – Method.

```
get_edge_data(datagraph, node_name1, node_name2, attribute_name)
get_edge_data(datagraph, edge, attribute_name)
```

Returns the value of attribute name on the edge between node_name1 and node_name2. edge is a tuple containing node_name1 and node_name2.

source

PlasmoData.get_edge_data – Method.

```
get_edge_data(datagraph, node_name1, node_name2, attribute_name)
get_edge_data(datagraph, edge, attribute_name)
```

Returns the value of attribute name on the edge between node_name1 and node_name2. edge is a tuple containing node_name1 and node_name2.

source

PlasmoData.get_graph_attributes – Method.

```
get_graph_attributes(dg::D) where {D <: DataGraphUnion}
```

Returns the list of attributes contained in the GraphData of dg

source

PlasmoData.get_graph_data – Method.

```
get_graph_data(dg::D) where {D <: DataGraphUnion}
```

Returns the data object from a DataGraph's or DataDiGraph's GraphData

source

PlasmoData.get_node_attributes – Method.

```
get_node_attributes(dg::D) where {D <: DataGraphUnion}
```

Returns the list of attributes contained in the NodeData of dg

source

PlasmoData.get_node_data – Method.

```
get_node_data(dg::D) where {D <: DataGraphUnion}
```

Returns the data object from a DataGraph's or DataDiGraph's NodeData

source

PlasmoData.get_node_data – Method.

```
get_node_data(dg::D, attribute::String) where {D <: DataGraphUnion}
```

Returns a vector of the edge data for the attribute

source

PlasmoData.get_node_data – Method.

```
get_node_data(dg::D, attribute_list; nodes = dg.nodes) where {D <: DataGraphUnion}
```

Returns a matrix of the node data for the attributes in attribute list in the order that the attributes are defined in the list. If nodes is defined, returns the subset of data corresponding to nodes in the order that they are defined in nodes

source

PlasmoData.get_node_data – Method.

```
get_node_data(datagraph, node_name, attribute_name)
```

Returns the value of attribute name on the given node

source

PlasmoData.get_node_data – Method.

```
get_node_data(datagraph, node_name, attribute_name)
```

Returns the value of attribute name on the given node

source

PlasmoData.get_ordered_edge_data – Method.

```
get_ordered_edge_data(dg::D) where {D <: DataGraphUnion}
```

Returns the ordered edge data matrix. For DataGraphs, this means all edges connected to dg.nodes[1] are ordered first, and so on. For DataDiGraphs, this means that all edges originating at dg.nodes[1] are ordered first, and so on for length(dg.nodes)

source

PlasmoData.get_ordered_edge_data – Method.

```
get_ordered_edge_data(dg::D, attribute::String) where {D <: DataGraphUnion}
```

Returns the ordered edge data vector for attribute. For DataGraphs, this means all edges connected to dg.nodes[1] are ordered first, and so on. For DataDiGraphs, this means that all edges originating at dg.nodes[1] are ordered first, and so on for length(dg.nodes)

source

PlasmoData.get_ordered_edge_data – Method.

```
get_ordered_edge_data(dg::D, attribute_list) where {D <: DataGraphUnion}
```

Returns the ordered edge data matrix for the attributes in attribute_list. For DataGraphs, this means all edges connected to dg.nodes[1] are ordered first, and so on. For DataDiGraphs, this means that all edges originating at dg.nodes[1] are ordered first, and so on for length(dg.nodes)

source

PlasmoData.get_path – Method.

```
get_path(datagraph, src_node, dst_node; algorithm = "Dijkstra")
```

Returns the shortest path in the `datagraph` between `src_node` and `dst_node`. Shortest path is computed by Dijkstra's algorithm

`algorithm` is a string key word. Options are limited to "Dijkstra", "BellmanFord"

source

`PlasmoData.get_path` – Method.

```
get_path(datagraph, src_node, intermediate_node, dst_node; algorithm = "Dijkstra")
```

Returns the shortest path in the `datagraph` between `src_node` and `dst_node` which passes through `intermediate node`.

`algorithm` is a string key word. Options are limited to "Dijkstra", "BellmanFord"

source

`PlasmoData.has_edge` – Method.

```
has_edge(datagraph, node1, node2)
```

Return `true` if there is an edge going from `node1` to `node2` in `datadigraph`. Else return false

source

`PlasmoData.has_edge` – Method.

```
has_edge(datagraph, node1, node2)
```

Return `true` if there is an edge between `node1` and `node2` in `datagraph`. Else return false

source

`PlasmoData.has_node` – Method.

```
has_node(datagraph, node)
has_node(datadigraph, node)
```

returns `true` if `node` is in the graph. Else return false

source

`PlasmoData.has_path` – Method.

```
has_path(datagraph, src_node, dst_node)
```

Returns true if a path exists in the `datagraph` between `src_node` to `dst_node`. Else returns false

source

`PlasmoData.has_path` – Method.

```
has_path(datagraph, src_node, intermediate_node, dst_node)
```

Returns true if a path exists in the `datagraph` between `src_node` and `dst_node` which passes through the `intermediate node`. Else returns false

source

`PlasmoData.index_to_nodes` – Method.

```
index_to_nodes(datagraph, index_list)
```

From a list of integer indeices, return a list of corresponding nodes in the `datagraph`

source

`PlasmoData.matrix_to_graph` – Method.

```
matrix_to_graph(matrix; diagonal = true, attribute="weight")
matrix_to_graph(array_3d; diagonal = true, attributes = ["weight$i" for i in
↪  1:size(array_3d)[3]])
```

Constructs a `DataGraph` object from a matrix and saves the matrix data as node attributes under the name `attribute`. If `diagonal = false`, the graph has a mesh structure, where each matrix entry is represented by a node, and each node is connected to the adjacent matrix entries/nodes. If `diagonal = true`, entries of the matrix are also connected to the nodes diagonal to them (i.e., entry (i,j) is connected to (i-1, j-1), (i + 1, j -1), etc.).

If a 3D matrix is passed to the function, it treats the first two dimensions as the matrix and then saves the data in the third dimension as different weights (i.e., for array of size dim1, dim2, and dim3, entry (i,j) has dim3 weights). `attribute_list` can be defined by the user to give names to each weight in the third dimension.

source

`PlasmoData.mvts_to_graph` – Method.

```
mvts_to_graph(mvts, attribute)
```

Converts a multivariate time series to a graph based on the covariance matrix. This first calculates the covariance of the multivariate time series (`mvts`) and then computes the covariance. It then forms the precision matrix by taking the inverse of the covariance and uses the function `symmetric_matrix_to_graph` to form the edge-weighted graph.

source

`PlasmoData.ne` – Method.

```
ne(dg::D) where {D <: DataGraphUnion}
```

Returns the number of edges in a DataGraph or DataDiGraph

source

`PlasmoData.nn` – Method.

```
nn(dg::D) where {D <: DataGraphUnion}
nv(dg::D) where {D <: DataGraphUnion}
```

Returns the number of nodes (vertices) in a DataGraph or DataDiGraph

source

`PlasmoData.nodes_to_index` – Method.

```
nodes_to_index(datagraph, node_list)
```

From a list of nodes in the `datagraph`, return a list of their corresponding integer indices

source

`PlasmoData.order_edges!` – Method.

```
order_edges!(dg) where {D <: DataGraphUnion}
```

Arranges in place the edges of `dg` so that they follow the order of `dg.nodes`. For `DataGraphs`, this means all edges connected to `dg.nodes[1]` are ordered first, and so on. For `DataDiGraphs`, this means that all edges originating at `dg.nodes[1]` are ordered first, and so on for `length(dg.nodes)`

source

`PlasmoData.remove_edge!` – Method.

```
remove_edge!(datadigraph, node1, node2)
remove_edge!(datadigraph, edge_tuple)
```

Remove the directed edge from node1 to node2 from the datadigraph.

source

PlasmoData.remove_edge! – Method.

```
remove_edge!(datagraph, node1, node2)
remove_edge!(datagraph, edge_tuple)
```

Remove the edge between node1 and node2 from the datagraph.

source

PlasmoData.remove_node! – Method.

```
remove_node!(datadigraph, node_name)
```

Removes the node (and any node data) from datadigraph

source

PlasmoData.remove_node! – Method.

```
remove_node!(datagraph, node_name)
```

Removes the node (and any node data) from datagraph

source

PlasmoData.rename_edge_attribute! – Method.

```
rename_edge_attribute!(dg::D, attribute, new_name) where {D <: DataGraphUnion}
```

Rename the edge data attribute as new_name. If attribute is not defined, returns an error.

source

PlasmoData.rename_graph_attribute! – Method.

```
rename_graph_attribute!(dg::D, attribute, new_name) where {D <: DataGraphUnion}
```

Rename the graph data attribute as new_name. If attribute is not defined, returns an error.

source

PlasmoData.rename_node_attribute! – Method.

```
rename_node_attribute!(dg::D, attribute, new_name) where {D <: DataGraphUnion}
```

Rename the node data attribute as new_name. If attribute is not defined, returns an error.

source

PlasmoData.run_EC_on_edges – Method.

```
run_EC_on_edges(dg, threshold_range; attribute = dg.edge_data.attributes[1], scale = false)
```

Returns the Euler Characteristic Curve by filtering the edges of the graph at each value in threshold_range and computing the Euler Characteristic after each filtration. If attribute is not defined, it defaults to the first attribute in the DataGraph's EdgeData. scale is a Boolean that indicates whether to scale the Euler Characteristic by the total number of objects (nodes + edges) in the original graph

source

PlasmoData.run_EC_on_nodes – Method.

```
run_EC_on_nodes(dg, threshold_range; attribute = dg.node_data.attributes[1], scale = false)
```

Returns the Euler Characteristic Curve by filtering the nodes of the graph at each value in threshold_range and computing the Euler Characteristic after each filtration. If attribute is not defined, it defaults to the first attribute in the DataGraph's NodeData. scale is a Boolean that indicates whether to scale the Euler Characteristic by the total number of objects (nodes + edges) in the original graph

source

PlasmoData.symmetric_matrix_to_graph – Method.

```
symmetric_matrix_to_graph(matrix; attribute="weight", tol = 1e-9)
```

Constructs a DataGraph object from a symmetric matrix and saves the values of the matrix to their corresponding edges. The resulting graph is fully connected (every node is connected to every node) and the number of nodes is equal to the dimension of the matrix. Matrix values are saved as edge weights under the name attribute. tol is the tolerance used when testing that the matrix is symmetric.

source

PlasmoData.tensor_to_graph – Method.

```
tensor_to_graph(tensor, attribute)
```

Constructs a graph from a 3-D array (a tensor). Each entry of the tensor is represented by a node, and each node is connected to the adjacent nodes in each dimension. This function creates the graph structure and saves the values of the tensor to their corresponding nodes as weight values under the name attribute.

source

PlasmoData.upstream_nodes – Method.

```
upstream_nodes(dg::DataDiGraph, node; algorithm = "bfs")
```

Return a list of all the nodes that are upstream of node in the DataDiGraph dg. Algorithm options are "bfs" and "dfs"

source