pyphysim Documentation

Darlan Cavalcante Moreira

Jul 28, 2020

CONTENTS

PyPhysim is a python library implementing functions, classes and simulators related to the physical layer of digital communications systems.

CHAPTER

ONE

LIST OF DOCUMENTATION ARTICLES

PyPhysim organization is described in *PyPhysim organization*, where a summary of the several packages in PyPhysim is shown. For instructions on how to implement new simulations with PyPhysim see *Implementing Simulators with PyPhysim*.

The complete list of articles in the PyPhysim documentation is shown below.

1.1 PyPhysim organization

The PyPhysim library is roughly organized in several packages with related modules. Most of the packages define functions and classes used to write physical layer simulators, but two packages are special: the *apps* package and the *tests* package. The *apps* package contains actual simulators that can be run, while the *tests* package, as the name suggests, have tests for the several packages in PyPhysim.

At last, there is also a *bin* directory containing a few scripts. One useful script to be run while developing PyPhysim is the **run_python_corerage** script, which will run the python-coverage program in all the test files in the tests folder. This will give a good estimate of the test coverage in PyPhysim (see *Writing Unittests for PyPhysim* for details about writting unittests for PyPhysim).

1.1.1 Packages in PyPhysim

A summary of the available packages in PyPhysim is shown below.

pyphysim package

Subpackages

pyphysim.c_extensions package

Module contents

Module containing compiled Cython extensions

pyphysim.cell package

Submodules

pyphysim.cell.cell module

Module that implements Cell and Cluster related classes.

class pyphysim.cell.cell.**AccessPoint**(*pos: complex*, *ap_id: Optional[Union[str, int]] = None*)

Bases: *pyphysim.cell.cell.Node*

Access point class.

An access point acts as a transmitter to one of more users.

Parameters

- **pos** (*complex*) – The central position of the cell in the complex grid.
- **ap_id** (*int, str, optional*) – The AccessPoint ID. If not provided the access point won't have an ID and its plot will shown a symbol at the access point location instead of the ID.

_plot_common_part(*ax: Any*) → None

Common code for plotting the classes. Each subclass must implement a *plot* method in which it calls the command to plot the class shape followed by _plot_common_part.

Parameters **ax** (*A matplotlib axis*) – The axis where the cell will be plotted.

add_user(*new_user:* pyphysim.cell.cell.Node, *relative_pos_bool: bool = True*) → None

Associate a new user with the access point.

Parameters

- **new_user** (Node) – The new user to be associated with the access point.
- **relative_pos_bool** (*bool*) – Indicates it the position of the *new_user* is relative.

delete_all_users() → None

Delete all users from the cell.

property num_users

Get method for the num_users property.

Returns The number of users associated with the AccessPoint.

Return type int

plot(*ax: Optional[Any] = None*) → None

Plot the AccessPoint using the matplotlib library.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the cell will be plotted. If not provided, a new figure (and axis) will be created.

property pos

Get the AccessPoint position.

Returns The AccessPoint position.

Return type complex

property users

Get method for the users property.

Returns The users associated with the AccessPoint.

Return type list[*Node*]

class pyphysim.cell.cell.**Cell**(*pos: complex*, *radius: float*, *cell_id: Optional[Union[str, int]] = None*, *rotation: float = 0.0*)

Bases: *pyphysim.cell.shapes.Hexagon*, *pyphysim.cell.cell.CellBase*

Class representing an hexagon cell.

Parameters

- **pos** (*complex*) – The central position of the cell in the complex grid.
- **radius** (*float*) – The cell radius.
- **cell_id** (*str, int, optional*) – The cell ID. If not provided the cell won't have an ID and its plot will shown a symbol in cell center instead of the cell ID.
- **rotation** (*float, optional*) – The rotation of the cell (regarding the cell center).

plot(*ax: Optional[Any] = None*) → None

Plot the cell using the matplotlib library.

If an axes 'ax' is specified, then the shape is added to that axis. Otherwise a new figure and axis are created and the shape is plotted to that.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the cell will be plotted. If not provided, a new figure (and axis) will be created.

class pyphysim.cell.cell.**Cell3Sec**(*pos: complex*, *radius: float*, *cell_id: Optional[Union[str, int]] = None*, *rotation: float = 0.0*)

Bases: *pyphysim.cell.cell.CellBase*

Class representing a cell with 3 sectors.

Each sector corresponds to an hexagon.

Parameters

- **pos** (*complex*) – The central position of the cell in the complex grid.
- **radius** (*float*) – The cell radius. The sector radius will be equal to half the cell radius.
- **cell_id** (*str, int, optional*) – The cell ID. If not provided the cell won't have an ID and its plot will shown a symbol in cell center instead of the cell ID.
- **rotation** (*float, optional*) – The rotation of the cell (regarding the cell center).

_calc_sectors_positions() → numpy.ndarray

Calculates the positions of the sectors with the current rotation, center position and radius.

Returns The positions of the 3 sectors.

Return type np.ndarray

_get_vertex_positions() → numpy.ndarray

Calculates the vertex positions ignoring any rotation and considering that the shape is at the origin (rotation and translation will be added automatically later).

Returns **vertex_positions** – The positions of the vertexes of the shape.

Return type np.ndarray

add_random_user_in_sector(*sector: int*, *user_color: Optional[str] = None*, *min_dist_ratio: float = 0.0*) → None

Adds a user randomly located in the specified *sector* of the cell.

Parameters

- **sector** (*int*) – The sector index. Can only be 1, 2 or 3.
- **user_color** (*str*) – Color of the user's marker.
- **min_dist_ratio** (*float*) – Minimum allowed (relative) distance between the cell center and the generated random user. The value must be between 0 and 0.7.

add_random_users_in_sector(*num_users: int*, *sector: int*, *user_color: Optional[str] = None*, *min_dist_ratio: float = 0.0*) → None

Add *num_users* users randomly in the specified *sector* of the cell

Parameters

- **num_users** (*int*) – Number of users to be added to the sector.
- **sector** (*int*) – The sector index. Can only be 1, 2 or 3.
- **user_color** (*str*) – Color of the user's marker.
- **min_dist_ratio** (*float*) – Minimum allowed (relative) distance between the cell center and the generated random user. The value must be between 0 and 0.7.

plot(*ax: Optional[Any] = None*) → None

Plot the cell using the matplotlib library.

If an axes 'ax' is specified, then the shape is added to that axis. Otherwise a new figure and axis are created and the shape is plotted to that.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the cell will be plotted. If not provided, a new figure (and axis) will be created.

property pos

Get the Cell3Sec position.

Returns The Cell3Sec position.

Return type complex

property radius

Get the radius of the Cell3Sec object.

Returns The radius of the Cell3Sec object.

Return type float

property rotation

Get method for the rotation property.

Returns The shape rotation.

Return type float

property secradius

Get method for the secradius property.

The radius of a sector.

Returns The radius of one sector of the Cell3Sec object.

Return type float

class pyphysim.cell.cell.**CellBase**(*pos: complex*, *radius: float*, *cell_id: Optional[Union[str, int]] = None*, *rotation: float = 0.0*, ***kw*)

Bases: *pyphysim.cell.shapes.Shape*, *pyphysim.cell.cell.AccessPoint*

Base class for all cell types.

A cell is an AccessPoint with a predefined shape, where the users associated with it are inside the shape.

Parameters

- **pos** (*complex*) – The central position of the cell in the complex grid.
- **radius** (*float*) – The cell radius.
- **cell_id** (*str, int, optional*) – The cell ID. If not provided the cell won't have an ID and its plot will shown a symbol in cell center instead of the cell ID.
- **rotation** (*float, optional*) – The rotation of the cell (regarding the cell center).

static _validate_ratio(*ratio: float*) → float

Return *ratio* if is valid, 1.0 if *ratio* is None, or throw an exception if it is not valid.

This is a helper method used in the add_border_user method implementation.

Parameters **ratio** (*float*) – The ratio (a number between 0 and 1).

Returns **ratio** – The valid ratio. If ratio parameter was 'None' then 1.0 will be returned.

Return type float

Raises **ValueError** – If *ratio* is not between 0 and 1.

add_border_user(*angles: Union[float, Iterable[float]], ratio: Optional[Union[float, Iterable[float]]] = None, user_color: Optional[Union[str, Iterable[str]]] = None*) → None

Adds a user at the border of the cell, located at a specified angle (in degrees).

If the *angles* variable is an iterable, one user will be added for each value in *angles*. Also, in that case ration and user_color may also be iterable with the same length of *angles* in order to specify individual ratio and user_color for each angle.

Parameters

- **angles** (*float | list[float] | np.ndarray*) – Angle(s) for which users will be added (may be a single number or an iterable).
- **ratio** (*float | list[float] | np.ndarray | None*) – The ration (relative distance from cell center) for which users will be added (may be a single number or an iterable). If not specified the users will be added to the cell's border at the angles specified in *angles*.
- **user_color** (*str | list[str], optional*) – Color of the user's marker.

Raises **ValueError** – If the ratio is invalid (negative or greater than 1).

add_random_user(*user_color: Optional[str] = None, min_dist_ratio: float = 0.0*) → None

Adds a user randomly located in the cell.

The variable *user_color* can be any color that the plot command and friends can understand. If not specified the default value of the node class will be used.

Parameters

- **user_color** (*str, optional*) – Color of the user's marker.
- **min_dist_ratio** (*float*) – Minimum allowed (relative) distance between the cell center and the generated random user. The value must be between 0 and 0.7.

add_random_users(*num_users: int, user_color: Optional[str] = None, min_dist_ratio: float = 0.0*) → None

Add *num_users* users randomly located in the cell.

Parameters

- **num_users** (*int*) – Number of users to be added to the cell.
- **user_color** (*str, optional*) – Color of the user's marker.
- **min_dist_ratio** (*float*) – Minimum allowed (relative) distance between the cell center and the generated random user. The value must be between 0 and 0.7.

add_user(*new_user:* pyphysim.cell.cell.Node, *relative_pos_bool: bool = True*) → None
Adds a new user to the cell.

Parameters

- **new_user** (Node) – The new user to be added to the cell.
- **relative_pos_bool** (*bool, optional (default to True)*) – Indicates if the 'pos' attribute of the *new_user* is relative to the center of the cell or not.

Returns

Return type None

Raises **ValueError** – If the user position is outside the cell (the user won't be added).

plot_border(*ax: Optional[Any] = None*) → None
Plot the border of the cell.

If an axes 'ax' is specified, then the shape is added to that axis. Otherwise a new figure and axis are created and the shape is plotted to that.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the cell will be plotted. If not provided, a new figure (and axis) will be created.

class pyphysim.cell.cell.**CellSquare**(*pos: complex, side_length: float, cell_id: Optional[Union[str, int]] = None, rotation: float = 0.0*)
Bases: *pyphysim.cell.shapes.Rectangle*, *pyphysim.cell.cell.CellBase*

Class representing a 'square' cell.

Parameters

- **pos** (*complex*) – The central position of the cell in the complex grid.
- **side_length** (*float*) – The cell side length.
- **cell_id** (*str, int, optional*) – The cell ID. If not provided the cell won't have an ID and its plot will shown a symbol in cell center instead of the cell ID.
- **rotation** (*float, optional*) – The rotation of the cell (regarding the cell center).

add_user(*new_user:* pyphysim.cell.cell.Node, *relative_pos_bool: bool = True*) → None
Adds a new user to the cell.

Parameters

- **new_user** (Node) – The new user to be added to the cell.
- **relative_pos_bool** (*bool, optional*) – Indicates if the 'pos' attribute of the *new_user* is relative to the center of the cell or not.

Raises **ValueError** – If the user position is outside the cell (the user won't be added).

plot(*ax: Optional[Any] = None*) → None
Plot the cell using the matplotlib library.

If an axes 'ax' is specified, then the shape is added to that axis. Otherwise a new figure and axis are created and the shape is plotted to that.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the cell will be plotted. If not provided, a new figure (and axis) will be created.

class pyphysim.cell.cell.**CellWrap**(*pos: complex*, *wrapped_cell:* pyphysim.cell.cell.CellBase, *include_users_bool: bool = False*)

Bases: *pyphysim.cell.cell.CellBase*

Class that wraps another cell.

Parameters

- **pos** (*complex*) – The central position where the wrapped cell will be in the complex grid.
- **wrapped_cell** (*T <= CellBase*) – The wrapped cell. It must be an object of some subclass of CellBase.
- **include_users_bool** (*bool*) – Set to True if the users of the original cells should appear in the wrapped version.

_get_vertex_positions() → numpy.ndarray

Calculates the vertex positions ignoring any rotation and considering that the shape is at the origin (rotation and translation will be added automatically later).

Returns **vertex_positions** – The positions of the vertexes of the shape.

Return type np.ndarray

property num_users

Get method for the num_users property.

Returns The number of users associated with the AccessPoint.

Return type int

plot(*ax: Optional[Any] = None*) → None

Plot the shape using the matplotlib library.

Parameters **ax** (*A matplotlib ax, optional*) – The ax where the shape will be plotted. If not provided, a new figure (and ax) will be created.

Notes

If an axes 'ax' is specified, then the shape is added to that axes. Otherwise a new figure and axes are created and the shape is plotted to that.

property radius

Get the radius of the CellWrap object.

Returns The radius of the CellWrap object.

Return type float

property rotation

Get the rotation of the CellWrap object.

Returns The rotation of the CellWrap object.

Return type float

property users

Get method for the users property.

Returns The users associated with the AccessPoint.

Return type list

class pyphysim.cell.cell.**Cluster**(*cell_radius: float*, *num_cells: int*, *pos: complex = 0j*, *cluster_id: Optional[int] = None*, *cell_type: str = 'simple'*, *rotation: float = 0.0*)

Bases: *pyphysim.cell.shapes.Shape*

Class representing a cluster of Hexagonal cells.

Valid cluster sizes are given by the formula $N = i^2 + i * j + j^2$ where i and j are integer numbers. The allowed values in the Cluster class are summarized below with the corresponding values of i and j.

i, j	N
1,0	01
1,1	03
2,0	04
2,1	07
3,1	13
3,2	19

Parameters

- **cell_radius** (*float*) – Radius of the cells in the cluster.
- **num_cells** (*int*) – Number of cells in the cluster.
- **pos** (*complex*) – Central Position of the Cluster in the complex grid.
- **cluster_id** (*int*) – ID of the cluster.
- **cell_type** (*str*) – The type of the cell as a string. It can be either 'simple', '3sec' or 'square'. If it is 'simple' it means the standard hexagon shaped cell. If '3sec' it means a 3 sectorized cell composed of 3 hexagons.
- **rotation** (*float*) – Rotation of the cluster.

static _calc_cell_height(*radius: float*) → float

Calculates the cell height from the cell radius.

Parameters **radius** (*float*) – The cell Radius.

Returns **height** – The cell height.

Return type float

static _calc_cell_positions(*cell_radius: float*, *num_cells: int*, *cell_type: str = 'simple'*, *rotation: Optional[float] = None*) → numpy.ndarray

Helper function used by the Cluster class.

The calc_cell_positions method calculates the position (and rotation) of the 'num_cells' different cells, each with radius equal to 'cell_radius', so that they properly fit in the cluster.

Parameters

- **cell_radius** (*float*) – Radius of each cell in the cluster.
- **num_cells** (*int*) – Number of cells in the cluster.
- **cell_type** (*str*) – The type of the cell. It should be a string with one of the possible values: 'simple', '3sec', or 'square'. If it is 'simple' it means the standard hexagon shaped cell. If '3sec' it means a 3 sectorized cell composed of 3 hexagons.
- **rotation** (*float | None, optional*) – Rotation of the cluster.

Returns **cell_positions** – The first column of *cell_positions* has the positions of the cells in a cluster with *num_cells* cells with radius *cell_radius*. The second column has the rotation of each cell.

Return type np.ndarray

static _calc_cell_positions_3sec(*cell_radius: float*, *num_cells: int*, *rotation: Optional[float] = None*) → numpy.ndarray

Helper function used by the Cluster class.

The _calc_cell_positions_3sec method calculates the position (and rotation) of the 'num_cells' different cells, each with radius equal to 'cell_radius', so that they properly fit in the cluster.

Parameters

- **cell_radius** (*float*) – Radius of each cell in the cluster.
- **num_cells** (*int*) – Number of cells in the cluster.
- **rotation** (*float | None, optional*) – Rotation of the cluster.

Returns **cell_positions** – The first column of *cell_positions* has the positions of the cells in a cluster with *num_cells* cells with radius *cell_radius*. The second column has the rotation of each cell.

Return type np.ndarray

static _calc_cell_positions_hexagon(*cell_radius: float*, *num_cells: int*, *rotation: Optional[float] = None*) → numpy.ndarray

Helper function used by the Cluster class.

The calc_cell_positions method calculates the position (and rotation) of the 'num_cells' different cells, each with radius equal to 'cell_radius', so that they properly fit in the cluster.

Parameters

- **cell_radius** (*float*) – Radius of each cell in the cluster.
- **num_cells** (*int*) – Number of cells in the cluster.
- **rotation** (*float | None, optional*) – Rotation of the cluster.

Returns The first column of *cell_positions* has the positions of the cells in a cluster with *num_cells* cells with radius *cell_radius*. The second column has the rotation of each cell.

Return type np.ndarray

static _calc_cell_positions_square(*side_length: float*, *num_cells: int*, *rotation: Optional[float] = None*) → numpy.ndarray

Helper function used by the Cluster class.

The calc_cell_positions method calculates the position (and rotation) of the 'num_cells' different cells, each with side equal to 'cell_radius', so that they properly fit in the cluster.

Parameters

- **side_length** (*float*) – The side length of each square cell in the cluster.
- **num_cells** (*int*) – Number of cells in the cluster.
- **rotation** (*float | None, optional*) – Rotation of the cluster.

Returns **cell_positions** – The first column of *cell_positions* has the positions of the cells in a cluster with *num_cells* cells with radius *cell_radius*. The second column has the rotation of each cell.

Return type np.ndarray

_calc_cluster_external_radius() → float
Calculates the cluster external radius.

The cluster external radius is equal to the radius of the smallest circle (located at the center of the cluster) that contains the cluster. This circle should touch only the most external vertexes of the cells in the cluster.

Get the vertex positions of the last cell.

Returns **external_radius** – The cluster external radius.

Return type float

static _calc_cluster_radius(*num_cells: int*, *cell_radius: float*) → float
Calculates the "cluster radius" for a cluster with "num_cells" cells, each cell with radius equal to "cell_radius". The cluster "radius" is equivalent to half the distance between two clusters when they are in a Grid.

Parameters

- **num_cells** (*int*) – Number of cells in the cluster.
- **cell_radius** (*float*) – Radius of each cell in the cluster.

Returns **cluster_radius** – The radius of a cluster with *num_cells* cells with radius *cell_radius*.

Return type float

Notes

The cluster "radius" is equivalent to half the distance between two clusters.

static _get_ii_and_jj(*num_cells: int*) → Tuple[int, int]
Valid cluster sizes are given by the formula

$$N = i^2 + i * j + j^2$$

where i and j are integer numbers and "N" is the number of cells in the cluster. This static function returns the values "i" and "j" for a given "N". The values are summarized below.

i, j	N
1,0	01
1,1	03
2,0	04
2,1	07
3,1	13
3,2	19

Parameters **num_cells** (*int*) – Number of cells in the cluster.

Returns **ii and jj** – The ii and jj values corresponding to number of cells 'num_cells'.

Return type (int,int)

Notes

If *num_cells* is not in the table above then (0, 0) will be returned.

_get_outer_vertexes(*vertexes: numpy.ndarray*, *central_pos: complex*, *distance: float*) → numpy.ndarray

Filter out vertexes closer to the shape center them *distance*.

This is a helper method used in the _get_vertex_positions method.

Parameters

- **vertexes** (*np.ndarray*) – The outer vertexes of the cluster.
- **central_pos** (*complex*) – Central position of the shape.
- **distance** (*float*) – A minimum distance. Any vertex that is closer to the shape center then this distance will be removed.

Returns **outer_vertexes** – The cluster outer vertexes.

Return type np.ndarray

_get_vertex_positions() → numpy.ndarray

Get the vertex positions of the cluster borders.

Returns **vertex_positions** – The vertex positions of the cluster borders.

Return type np.ndarray

Notes

This is only valid for cluster sizes from 1 to 19.

_ii_and_jj = {1: (1, 0), 3: (1, 1), 4: (2, 0), 7: (2, 1), 13: (3, 1), 19: (3, 2)

_normalized_cell_positions: Dict[int, numpy.ndarray] = {}

add_border_users(*cell_ids: Union[Iterable[int], int]*, *angles: Union[Iterable[float], float]*, *ratios: Union[Iterable[float], float] = 1.0*, *user_color: Optional[Union[str, Iterable[str]]] = None*) → None

Add users to all the cells indicated by *cell_indexes* at the specified angle(s) (in degrees) and ratio (relative distance from the center to the border of the cell).

Parameters

- **cell_ids** (*int | list[int] | np.ndarray*) – IDs of the cells in the Cluster for which users will be added. The first cell has an ID equal to 1 and *cell_ids* may be an iterable with the IDs of several cells.
- **angles** (*float | list[float] | np.ndarray*) – Angles (in degrees)
- **ratios** (*float | list[float]*) – Ratios (from 0 to 1)
- **user_color** (*str | list[str]*) – Color of the user's marker.

Examples

```
>>> cluster = Cluster(cell_radius=1.0, num_cells=3)
>>> # Add a single user in the angle of 30 degrees with a ration of
>>> #  0.9 to the first cell in the cluster
>>> cluster.add_border_users(1, 30, 0.9)
>>>
>>> # Add 3 users at the angles of 0, 95 and 185 degrees to the
>>> # second cell of the cluster
>>> cluster.add_border_users(2, [0, 95, 185], 0.9, 'b')
>>>
>>> # Add one user in each cell at the angle of 10 degrees
>>> cluster.add_border_users([1, 2, 3], 10, 0.9, 'g')
>>>
>>> # Add a user in each cell at different angles per cell
>>> cluster.add_border_users([1, 2, 3], [90, 150, 190], 0.9, 'y')
>>>
>>> # Add multiple users to multiple cells at different angles
>>> cluster.add_border_users(                [1, 2, 3], [[180, 270], [-30],␣
↪[60, 120]], 0.9, 'k')
```

add_random_users(*cell_ids: Optional[Union[Iterable[int], int]] = None, num_users: Union[Iterable[int], int] = 1, user_color: Optional[Union[str, Iterable[str]]] = None, min_dist_ratio: Union[Iterable[float], float] = 0.0*) → None

Adds one or more users to the Cells with the specified cell IDs (the first cell has an ID equal to 1.).

Parameters

- **cell_ids** (*int | list[int] | np.ndarray*) – IDs of the cells in the Cluster for which users will be added. The first cell has an ID equal to 1 and *cell_ids* may be an iterable with the IDs of several cells. If not provided, all cells will be assumed.
- **num_users** (*int | list[int] | np.ndarray*) – Number of users to be added to each cell.
- **user_color** (*str | list[str], optional*) – Color of the user's marker.
- **min_dist_ratio** (*float, list[float], optional*) – Minimum allowed (relative) distance between the cell center and the generated random user. See Cell.add_random_user method for details.

Notes

If *cell_ids* is an iterable then the other attributes (num_users, user_color and min_dist_ratio) may also be iterable with the same length of cell_ids in order to specifying individual values for each cell ID.

calc_dist_all_users_to_each_cell() → numpy.ndarray

Returns a matrix with the distance from each user to each cell center.

This matrix is suitable to later calculate the path loss from each base station to each mobile station.

Because usually the base station is the transmitter and the mobile station is the receiver the matrix is such that each column corresponds to a different base station and each row corresponds to a different mobile station.

Returns **all_dists** – Distance from each cell center to each user.

Return type np.ndarray

Notes

There is no explicit indication from which cell each user came from. However, in a case, for instance, where there are 3 cells in the cluster with 2, 2 and 3 users in each of them, respectively, then the first 2 rows correspond to the users in the first cell, the following 2 rows correspond to the users in the second cell and the last three rows correspond to the users in the third cell.

calc_dist_all_users_to_each_cell_no_wrap_around() → numpy.ndarray
Returns a matrix with the distance from each user to each cell center.

This matrix is suitable to later calculate the path loss from each base station to each mobile station.

Because usually the base station is the transmitter and the mobile station is the receiver the matrix is such that each column corresponds to a different base station and each row corresponds to a different mobile station.

Returns **all_dists** – Distance from each cell center to each user.

Return type np.ndarray

Notes

There is no explicit indication from which cell each user came from. However, in a case, for instance, where there are 3 cells in the cluster with 2, 2 and 3 users in each of them, respectively, then the first 2 rows correspond to the users in the first cell, the following 2 rows correspond to the users in the second cell and the last three rows correspond to the users in the third cell.

calc_dists_between_cells() → numpy.ndarray
This method calculates the distance between any two cells in the cluster possibly considering wrap around.

If the *create_wrap_around_cells* method was called before this one, then when calculating the distance between two cells if the distance between a given cell and the wrapped version of another cell is smaller then the distance to that other cell it will be sued instead.

For instance, the

Returns **dists** – A matrix with the distance from each cell to each other cell in the cluster.

Return type np.ndarray

property cell_height
Get method for the cell_height property.

Returns The height of the cells in the Cluster.

Return type float

property cell_id_fontsize
Get method for the cell_id_fontsize property.

The value of cell_id_fontsize only matters for plotting the cluster.

Returns The font size that should be used for the cell IDs in the plot.

Return type int | None

property cell_radius
Get method for the cell_radius property.

Returns The radius of the cells in the Cluster.

Return type float

create_wrap_around_cells(*include_users_bool: bool = False*) → None
This function will create the wrapped cells, as well as the wrap info data.

Parameters **include_users_bool** (*bool*) – Set to True if the users of the original cells should appear in the wrapped version.

delete_all_users(*cell_id: Optional[Union[Iterable[int], int]] = None*) → None
Remove all users from one or more cells.

If cell_id is an integer > 0, only the users from the cell whose index is *cell_id* will be removed. If cell_id is an iterable, then the users of cells pointed by it will be removed. If cell_id is *None* or not specified, then the users of all cells will be removed.

Parameters **cell_id** (*int | list[int], optional*) – ID(s) of the cells from which users will be removed. If equal to None, all the users from all cells will be removed.

property external_radius
Get the external_radius of the Cluster.

Returns The external_radius of the Cluster.

Return type float

get_all_users() → List[*pyphysim.cell.cell.Node*]
Return all users in the cluster.

Returns **all_users** – A list with all users in the cluster.

Return type list[*Node*]

get_cell_by_id(*cell_id: int*) → *pyphysim.cell.cell.CellBase*
Get the cell in the Cluster with the given *cell_id*.

Parameters **cell_id** (*int*) – The ID of the desired cell.

Returns **c** – The desired cell.

Return type *Cell*

property num_cells
Get method for the num_cells property.

Returns Number of cells in the Cluster.

Return type int

property num_users
Get method for the num_users property.

Returns The number of users in the Cluster.

Return type int

plot(*ax: Optional[Any] = None*) → None
Plot the cluster.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the cluster will be plotted. If not provided, a new figure (and axis) will be created.

plot_border(*ax: Optional[Any] = None*) → None
Plot only the border of the Cluster.

Only work's for cluster sizes that can calculate the cluster vertices, such as cluster with 1, 7 or 19 cells.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the cluster will be plotted. If not provided, a new figure (and axis) will be created.

property pos
Get the Cluster position.

Returns The Cluster position.

Return type complex

property radius
Get the radius of the Cluster object.

Returns The radius of the Cluster object.

Return type float

property rotation
Get method for the rotation property.

Returns The shape rotation.

Return type float

property vertices
Get the vertex positions of the cluster borders.

Returns **vertex_positions** – The vertex positions of the cluster borders.

Return type np.ndarray

Notes

This is only valid for cluster sizes from 1 to 19.

class pyphysim.cell.cell.**Grid**
Bases: object

Class representing a grid of clusters of cells or a single cluster with its surrounding cells.

Valid cluster sizes are given by the formula $N = i^2 + i * j + j^2$ where i and j are integer numbers. The values allowed in the Cluster are summarized below with the corresponding values of i and j.

i, j	N
1,0	01
1,1	03
2,0	04
2,1	07
3,1	13
3,2	19

_calc_cluster_pos2() → complex
Calculates the central position of clusters with 2 cells.

Returns **central_pos** – Central position of the next cluster to be added to the Grid.

Return type complex

Notes

The returned central position will depend on how many clusters were already added to the grid.

_calc_cluster_pos3() → complex
Calculates the central position of clusters with 3 cells.

Returns **central_pos** – Central position of the next cluster to be added to the Grid.

Return type complex

Notes

The returned central position will depend on how many clusters were already added to the grid.

_calc_cluster_pos7() → complex
Calculates the central position of clusters with 7 cells.

Returns **central_pos** – Central position of the next cluster to be added to the Grid.

Return type complex

Notes

The returned central position will depend on how many clusters were already added to the grid.

_colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'w']

_repr_png_() → Any
Return the PNG representation of the shape.

_repr_some_format_(*extension: str = 'png'*, *axis_option: str = 'equal'*) → Any
Return the representation of the shape in the desired format.

Parameters

- **extension** (*str*) – The extension of the desired format. This should be something that the savefig method in a matplotlib figure can understand, such as 'png', 'svg', stc.
- **axis_option** (*str*) – Option to be given to the ax.axis function.

Returns The representation in the desired format.

Return type Any

_repr_svg_() → Any
Return the SVG representation of the shape.

clear() → None
Clear everything in the grid.

create_clusters(*num_clusters: int*, *num_cells: int*, *cell_radius: float*) → None
Create the clusters in the grid.

Parameters

- **num_clusters** (*int*) – Number of clusters to be created in the grid.
- **num_cells** (*int*) – Number of cells per clusters.
- **cell_radius** (*float*) – The radius of each cell.

get_cluster_from_index(*index: int*) → *pyphysim.cell.cell.Cluster*
Return the cluster object with index *index* in the Grid.

Parameters **index** (*int*) – The index of the desirable cluster.

Returns The desired cluster in the Grid.

Return type *Cluster*

property num_clusters
Get method for the num_clusters property.

Returns The number of clusters in teh grid.

Return type int

plot(*ax: Optional[Any] = None*) → None
Plot the grid of clusters.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the grid will be plotted. If not provided, a new figure (and axis) will be created.

class pyphysim.cell.cell.**Node**(*pos: complex*, *plot_marker: str = '*'*, *marker_color: str = 'r'*, *cell_id: Optional[Union[str, int]] = None*, *parent_pos: Optional[complex] = None*)
Bases: *pyphysim.cell.shapes.Coordinate*

Class representing a node in the network.

Parameters

- **pos** (*complex*) – The position of the node in the complex grid.
- **plot_marker** (*str*) – The marker to be used in a plot to represent the Node. This marker should be something that matplotlib can understand, such as '*', for instance.
- **marker_color** (*str*) – The color that will be used to plot the marker representing the Node. This color should be something that matplotlib can understand, such as 'r' for the color red, for instance.
- **cell_id** (*str, int, optional*) – The ID of the cell where the Node is located.
- **parent_pos** (*complex*) – The position of the cell where the Node is located (if any).

plot_node(*ax: Optional[Any] = None*) → None
Plot the node using the matplotlib library.

If an axes 'ax' is specified, then the node is added to that axes. Otherwise a new figure and axes are created and the node is plotted to that.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the node will be plotted. If not provided, a new figure (and axis) will be created.

property relative_pos
Get method for the relative_pos property.

Returns The relative position of the Node regarding its parent Node's position.

Return type complex | None

pyphysim.cell.shapes module

Module implementing geometric shapes.

Each shape knows how to plot itself.

class pyphysim.cell.shapes.**Circle**(*pos: complex*, *radius: float*)
Bases: *pyphysim.cell.shapes.Shape*

Circle shape class.

A circle is initialized only from a coordinate and a radius.

Parameters

- **pos** (*complex*) – Coordinate of the center of the circle.
- **radius** (*float*) – Circle's radius.

_get_vertex_positions() → numpy.ndarray
Calculates the vertex positions considering that the circle is at the origin (translation will be added automatically later).

Returns **vertex_positions** – The positions of the vertexes of the shape.

Return type np.ndarray

Notes

It does not make much sense to get the vertexes of a circle, since a circle 'has' infinite vertexes. However, for consistence with the Shape's class interface the _get_vertex_positions is implemented such that it returns a subset of the circle vertexes. The number of returned vertexes was arbitrarily chosen as 12.

get_border_point(*angle: float*, *ratio: Optional[float] = None*) → complex
Calculates the coordinate of the point that intercepts the border of the circle if we go from the origin with a given angle (in degrees).

Parameters

- **angle** (*float*) – Angle in degrees
- **ratio** (*float*) – The ratio from the cell center to the border where the desired point is located. It must be a value between 0 and 1.

Returns **point** – A point in the line between the circle's center and the circle's border with the desired angle. If ratio is equal to one the point will be in the end of the line (touching the circle's border)

Return type complex

is_point_inside_shape(*point: complex*) → bool
Test is a point is inside the circle

Parameters **point** – A single complex number.

Returns True if *point* is inside the circle, False otherwise.

Return type inside_or_not

plot(*ax: Any = None*) → None
Plot the circle using the Matplotlib library.

Parameters **ax** (*A matplotlib axis, optional*) – The axis where the shape will be plotted. If not provided, a new figure (and axis) will be created.

Notes

If an axes 'ax' is specified, then the shape is added to that axes. Otherwise a new figure and axes are created and the shape is plotted to that.

class pyphysim.cell.shapes.**Coordinate**(*pos: complex*)
Bases: object

Base class for a coordinate in a 2D grid.

A Coordinate object knows its location in the grid (represented as a complex number) and how to calculate the distance from it to another location.

calc_dist(*other:* pyphysim.cell.shapes.Coordinate) → float
Calculates the distance to another coordinate.

Parameters **other** (Coordinate) – A different coordinate object.

Returns **dist** – Distance from self to the other coordinate.

Return type float

move_by_relative_coordinate(*rel_pos: complex*) → None
Move from the current position to the relative coordinate.

This is equivalent to moving to a new position given by the current position plus *coordinate*.

Parameters **rel_pos** (*complex*) – Relative coordinate

move_by_relative_polar_coordinate(*radius: float*, *angle: float*) → None
Move from the current position to the relative coordinate.

This is equivalent to moving to a new position given by the current position plus a the provided coordinate.

Parameters

- **radius** (*float*) – Distance of the movement in the direction given by *angle*.
- **angle** (*float*) – Angle (in radians) pointing the direction of the movement.

property pos
Get the coordinate position as a complex number.

Returns The coordinate position (a complex number).

Return type complex

class pyphysim.cell.shapes.**Hexagon**(*pos: complex*, *radius: float*, *rotation: float = 0*, ***kw*)
Bases: *pyphysim.cell.shapes.Shape*

Hexagon shape class.

Besides the *pos*, *radius* and *rotation* properties from the Shape base class, the Hexagon also has a height property (read-only) from the base of the Hexagon to its center.

Parameters

- **pos** (*complex*) – Coordinate of the shape in the complex grid.
- **radius** (*float*) – Radius of the hexagon. It must be a positive number.
- **rotation** (*float*) – Rotation of the hexagon in degrees.

_get_vertex_positions() → numpy.ndarray
Calculates the vertex positions ignoring any rotation and considering that the hexagon is at the origin (rotation and translation will be added automatically later).

Returns **vertex_positions** – The positions of the vertexes of the shape.

Return type np.ndarray

property height
Get method for the height property.

Returns The height of the Hexagon.

Return type float

class pyphysim.cell.shapes.**Rectangle**(*first: complex*, *second: complex*, *rotation: float = 0*, ***kw*)
Bases: *pyphysim.cell.shapes.Shape*

Rectangle shape class.

The rectangle is initialized from two coordinates as well as from the rotation.

Parameters

- **first** (*complex*) – First coordinate (without rotation).
- **second** (*complex*) – Second coordinate (without rotation).
- **rotation** (*float*) – Rotation of the rectangle in degrees.

_get_vertex_positions() → numpy.ndarray
Calculates the vertex positions ignoring any rotation and considering that the rectangle is at the origin (rotation and translation will be added automatically later).

Returns **vertex_positions** – The positions of the vertexes of the shape.

Return type np.ndarray

_repr_some_format_(*extension: str = 'png'*, *axis_option: str = 'tight'*) → Any
Return the representation of the shape in the desired format.

Parameters

- **extension** (*str*) – The extension of the desired format. This should be something that the savefig method in a matplotlib figure can understand, such as 'png', 'svg', stc.
- **axis_option** (*str*) – Option to be given to the ax.axis function.

Notes

We only subclass the _repr_some_format_ method from the Shape class here so that we can change the axis_option to 'tight' (default in the Shape class is 'equal').

is_point_inside_shape(*point: complex*) → bool
Test is a point is inside the rectangle

Parameters **point** (*complex*) – A single complex number.

Returns True if *point* is inside the rectangle, False otherwise.

Return type bool

class pyphysim.cell.shapes.**Shape**(*pos: complex*, *radius: float*, *rotation: float = 0*, ***kw*)
Bases: *pyphysim.cell.shapes.Coordinate*

Base class for all 2D shapes.

Each subclass must implement the _get_vertex_positions method.

Parameters

- **pos** (*complex*) – Coordinate of the shape in the complex grid.
- **radius** (*float*) – Radius of the shape. It must be positive.
- **rotation** (*float*) – Rotation of the shape in degrees.

abstract _get_vertex_positions() → numpy.ndarray

Calculates the vertex positions ignoring any rotation and considering that the shape is at the origin (rotation and translation will be added automatically later).

Returns **vertex_positions** – The positions of the vertexes of the shape (as a 1D numpy array).

Return type np.ndarray

Notes

Not implemented. Must be implemented in a subclass and return a one-dimensional numpy array (complex dtype) with the vertex positions.

_repr_png_() → Any

Return the PNG representation of the shape.

_repr_some_format_(*extension: str = 'png'*, *axis_option: str = 'equal'*) → Any

Return the representation of the shape in the desired format.

Parameters

- **extension** (*str*) – The extension of the desired format. This should be something that the savefig method in a matplotlib figure can understand, such as 'png', 'svg', stc.
- **axis_option** (*str*) – Option to be given to the ax.axis function.

Returns

Return type output

_repr_svg_() → Any

Return the SVG representation of the shape.

static calc_rotated_pos(*cur_pos: ComplexOrArray*, *angle: float*) → ComplexOrArray

Rotate the complex numbers in the *cur_pos* array by *angle* (in degrees) around the origin.

Parameters

- **cur_pos** (*complex | np.ndarray*) – The complex number(s) to be rotated.
- **angle** (*float*) – Angle in degrees to rotate the positions.

Returns **rotated_pos** – The rotate complex number(s).

Return type complex | np.ndarray

get_border_point(*angle: float*, *ratio: Optional[float] = None*) → complex

Calculates the coordinate of the point that intercepts the border of the shape if we go from the origin with a given angle (in degrees).

Parameters

- **angle** (*float*) – Angle in degrees.
- **ratio** (*float*) – The ratio from the cell center to the border where the desired point is located. This MUST be a value between 0 and 1.

Returns **point** – A point in the line between the shape's center and the shape's border with the desired angle. If ratio is equal to one the point will be in the end of the line (touching the shape's border)

Return type complex

is_point_inside_shape(*point: complex*) → bool
Test is a point is inside the shape.

Parameters **point** – A single complex number.

Returns True if *point* is inside the shape, False otherwise.

Return type inside_or_not

plot(*ax: Any = None*) → None
Plot the shape using the matplotlib library.

Parameters **ax** (*A matplotlib ax, optional*) – The ax where the shape will be plotted. If not provided, a new figure (and ax) will be created.

Notes

If an axes 'ax' is specified, then the shape is added to that axes. Otherwise a new figure and axes are created and the shape is plotted to that.

property radius
Get method for the radius property.

Returns The Shape radius.

Return type float

property rotation
Get method for the rotation property.

Returns The shape rotation.

Return type float

property vertices
Get method for the vertices property.

Returns The shape vertexes.

Return type np.ndarray

property vertices_no_trans_no_rotation
Get the shape vertexes without translation and rotation.

Returns **vertex_positions** – The positions of the vertexes of the shape without any translation or rotation (as a 1D numpy array).

Return type np.ndarray

Module contents

Package with cell and shapes related modules.

pyphysim.channels package

Submodules

pyphysim.channels.antennagain module

class pyphysim.channels.antennagain.**AntGainBS3GPP25996**(*number_of_sectors: int = 3*)

Bases: *pyphysim.channels.antennagain.AntGainBase*

Class for antenna model defined by 3GPP in the 25996 norm for sectorized Base Stations.

The antenna gain (in dBi) will depend on the number of sectors of the Base Station.

NOTE: The antenna pattern here is targeted for diversity-oriented implementations (i.e. large inter-element spacings). For beamforming applications that require small spacings, alternative antenna designs may have to be considered leading to a different antenna pattern.

Parameters **number_of_sectors** (*int*) – The number of sectors of the base station. It can be either 3 or 6.

get_antenna_gain(*angle: NumberOrArray*) → NumberOrArray

Get the antenna gain for the given angle.

Parameters **angle** (*float | np.ndarray*) – Angle between the direction of interest and the boresight of the antenna. This can also be a numpy array with angles.

Returns The gain (in linear scale) for the provided angle.

Return type float | np.ndarray

class pyphysim.channels.antennagain.**AntGainBase**

Bases: object

Base class for antenna models.

get_antenna_gain(*angle: NumberOrArray*) → NumberOrArray

Get the antenna gain for the given angle.

Parameters **angle** (*float | np.ndarray*) – Angle between the direction of interest and the boresight of the antenna. This can also be a numpy array with angles.

Returns The gain (in linear scale) for the provided angle.

Return type float | np.ndarray

class pyphysim.channels.antennagain.**AntGainOmni**(*ant_gain: Optional[float] = None*)

Bases: *pyphysim.channels.antennagain.AntGainBase*

Class for Omnidirectional antenna gain model.

Parameters **ant_gain** (*float, optional*) – The antenna gain (in dBi). If not provided then 0dBi will be assumed.

get_antenna_gain(*angle: NumberOrArray*) → NumberOrArray

Get the antenna gain for the given angle.

Parameters **angle** (*float | np.ndarray*) – Angle between the direction of interest and the boresight of the antenna. This can also be a numpy array with angles.

Returns The gain (in linear scale) for the provided angle.

Return type float | np.ndarray

pyphysim.channels.fading module

class pyphysim.channels.fading.**TdlChannel**(*fading_generator: Union[pyphysim.channels.fading_generators.JakesSampleGenerator, pyphysim.channels.fading_generators.RayleighSampleGenerator], channel_profile: Optional[pyphysim.channels.fading.TdlChannelProfile] = None, tap_powers_dB: Optional[numpy.ndarray] = None, tap_delays: Optional[numpy.ndarray] = None, Ts: Optional[float] = None*)

Bases: object

Tapped Delay Line channel model, which corresponds to a multipath channel.

You can create a new TdlChannel object either specifying the channel profile or specifying both the channel tap powers and delays.

Parameters

- **fading_generator** (*FadingGenerator*) – The instance of a fading generator in the *fading_generators* module. It should be a subclass of FadingSampleGenerator. The fading generator will be used to generate the channel samples. If the shape of the fading_generator is not None, then it must contain two positive integers, and a MIMO transmission will be employed, where the first integer in shape corresponds to the number of receive antennas while the second integer corresponds to the number of transmit antennas
- **channel_profile** (TdlChannelProfile) – The channel profile, which specifies the tap powers and delays.
- **tap_powers_dB** (*np.ndarray*) – The powers of each tap (in dB). Dimension: *L x 1* Note: The power of each tap will be a negative number (in dB).
- **tap_delays** (*np.ndarray*) – The delay of each tap (in seconds). Dimension: *L x 1*

_TdlChannel__prepare_transmit_signal_shape(*signal: numpy.ndarray*) → numpy.ndarray

Helper method called in corrupt_data and corrupt_data_in_freq_domain methods to prepare the shape of transmit *signal*.

It there is only one transmit antenna but signal is 1D, then an extra dimension will be added to *signal*. Otherwise the *signal* will be just returned.

Parameters **signal** (*np.ndarray*) – The signal to be transmitted. This should be 1D for SISO systems (or SIMO systems) and 2D for MIMO systems.

Returns Either the same signal of signal with an added dimension.

Return type np.ndarray

_set_fading_generator_shape(*new_shape: Optional[Tuple[int, ...]]*) → None

Set the shape of the fading generator.

Parameters **new_shape** (*tuple[int], None*) – The new shape of the fading generator. Note that the actual shape will be set to (self.num_taps, new_shape)

property channel_profile
Return the channel profile.

Returns The channel profile.

Return type *TdlChannelProfile*

corrupt_data(*signal: numpy.ndarray*) → numpy.ndarray
Transmit the signal though the TDL channel.

Parameters **signal** (*np.ndarray*) – The signal to be transmitted. This should be 1D for SISO systems (or SIMO systems) and 2D for MIMO systems.

Returns The received signal after transmission through the TDL channel

Return type np.ndarray

corrupt_data_in_freq_domain(*signal: numpy.ndarray*, *fft_size: int*, *carrier_indexes: Optional[Union[numpy.ndarray, List[int], slice]] = None*) → numpy.ndarray
Transmit the signal through the TDL channel, but in the frequency domain.

This is ROUGHLY equivalent to modulating *signal* with OFDM using *fft_size* subcarriers, transmitting through a regular TdlChannel, and then demodulating with OFDM to recover the received signal.

One important difference is that here the channel is considered constant during the transmission of *fft_size* elements in *signal*, and then it is varied by the equivalent of the variation for that number of elements. That is, the channel is block static.

Parameters

- **signal** (*np.ndarray*) – The signal to be transmitted. This should be 1D for SISO systems (or SIMO systems) and 2D for MIMO systems.
- **fft_size** (*int*) – The size of the Fourier transform to get the frequency response.
- **carrier_indexes** (*slice | np.ndarray | list[int]*) – The indexes of the subcarriers where signal is to be transmitted. If it is None assume all subcarriers will be used. This can be a slice object or a numpy array of integers.

Returns The received signal after transmission through the TDL channel

Return type np.ndarray

generate_impulse_response(*num_samples: int = 1*) → None
Generate a new impulse response of all discretized taps (not including possible zero padding) for *num_samples* channel realizations.

NOTE: This method is automatically called in the *corrupt_data* and *corrupt_data_in_freq_domain* methods and you don't need to call it befortransmitting data. After one of them has been called the generated impulse response can ge get with the *get_last_impulse_response* method.

The number of discretized taps of the generated impulse response will depend on the channel delay profile (the tap_delays passed during creation of the TdlChannel object) as well as on the sampling interval.

As an example, the COST259 TU channel profile has 20 different taps where the last one has a delay equal to 2.14 microseconds. If the sampling interval is configured as 3.25e-08 then the discretized channel will have more than 60 taps (including the zeros padding), where only 15 taps are different from zero. These 15 taps are what is returned by this method.

Alternatively, with a sampling time of 1e-6 you will end up with only 3 discretized taps.

Parameters **num_samples** (*int*) – The number of samples to generate (for each tap).

get_last_impulse_response() → *pyphysim.channels.fading.TdlImpulseResponse*
Get the last generated impulse response.

A new impulse response is generated when the method *corrupt_data* is called. You can use the *get_last_impulse_response* method to get the impulse response used to corrupt the last data.

Returns The impulse response of the channel that was used to corrupt the last data.

Return type *TdlImpulseResponse*

property num_rx_antennas
Get the number of receive antennas.

Returns The number of receive antennas.

Return type int

property num_taps
Number of taps not including zero taps after discretization.

Returns The number of taps (not including padding)

Return type int

property num_taps_with_padding
Number of taps including zero taps after discretization.

Returns The number of taps (including padding)

Return type int

property num_tx_antennas
Get the number of transmit antennas.

Returns The number of transmit antennas.

Return type int

set_num_antennas(*num_rx_antennas: int*, *num_tx_antennas: int*) → None
Set the number of transmit and receive antennas for MIMO transmission.

Set both *num_rx_antennas* and *num_tx_antennas* to None for SISO transmission

Parameters

- **num_rx_antennas** (*int*) – The number of receive antennas.
- **num_tx_antennas** (*int*) – The number of transmit antennas.

property switched_direction
Get the value of *switched_direction*.

Returns True if direction is switched and False otherwise.

Return type bool

class pyphysim.channels.fading.**TdlChannelProfile**(*tap_powers_dB: Optional[numpy.ndarray] = None*, *tap_delays: Optional[numpy.ndarray] = None*, *name: str = 'custom'*)

Bases: object

Channel Profile class.

This class is just a nice way to present known profiles from the norm or the literature, which are represented as instances of this class.

A TDL channel profile store information about the TDL taps. That is, it stores the power and delay of each tap. The power and delay of each tap can be accessed through the *tap_powers_** and *tap_delays* properties.

Some profiles are defined as objects of this class, such as `COST259_TUx`, `COST259_RAx` and `COST259_HTx`. These can be used when instantiating a *TdlChannel* object.

Note that the tap powers and delays are not necessarily *discretized* to some sampling interval.

Parameters

- **tap_powers_dB** (*np.ndarray*) – The tap powers (in dB). If both *tap_powers_dB* and *tap_delays* are None then a single tap with 0dB power will be assumed at delay 0.
- **tap_delays** (*np.ndarray*) – The tap delays.
- **name** (*str*) – A name for the channel profile

Examples

```
>>> jakes_generator = fading_generators.JakesSampleGenerator(Ts=3.25e-8)
>>> tdlchannel = TdlChannel(jakes_generator, channel_profile=COST259_TUx)
```

property Ts

Get the sampling interval used for discretizing this channel profile object.

If it is not discretized then this returns None.

Returns The sampling interval (in seconds).

Return type float, None

_calc_discretized_tap_powers_and_delays(*Ts: float*) → Tuple[numpy.ndarray, numpy.ndarray]

Discretize the taps according to the sampling time.

The discretized taps will be equally spaced and the delta time from two taps corresponds to the sampling time.

Parameters **Ts** (*float*) – The sampling time.

Returns A tuple with the discretized powers and delays.

Return type np.ndarray, np.ndarray

get_discretize_profile(*Ts: float*) → *pyphysim.channels.fading.TdlChannelProfile*

Compute the discretized taps (power and delay) and return a new discretized TdlChannelProfile object.

The tap powers and delays of the returned TdlChannelProfile object correspond to the taps and delays of the TdlChannelProfile object used to call *get_discretize_profile* after discretizing with the sampling interval *Ts*.

Parameters **Ts** (*float*) – The sampling time for the discretization of the tap powers and delays.

Returns The discretized channel profile

Return type *TdlChannelProfile*

property is_discretized

Returns True if the channel profile is discretized

property mean_excess_delay
The mean excess delay is the first moment of the power delay profile and is defined to be

$$\overline{\tau} = \frac{\sum_k P(\tau_k)\tau_k}{\sum_k P(\tau_k)}$$

Returns The mean excess delay.

Return type float

property name
Get the profile name.

Returns Profile name.

Return type str

property num_taps
Get the number of taps in the profile.

Returns Number of taps before discretization (does not count possible padding).

Return type int

property num_taps_with_padding
Get the number of taps in the profile including zero-padding when the profile is discretized.

If the profile is not discretized an exception is raised.

Returns Number of taps after discretization (it counts possible any added padding).

Return type int

property rms_delay_spread
The RMS delay spread is the square root of the second central moment of the power delay profile. It is defined to be

$$\sigma_t = \sqrt{\overline{t^2} - \overline{\tau}^2}$$

where

$$\overline{\tau^2} = \frac{\sum_k P(\tau_k)\tau_k^2}{\sum_k P(\tau_k)}$$

Typically, when the symbol time period is greater than 10 times the RMS delay spread, no ISI equalizer is needed in the receiver.

Returns The RMS delay spread.

Return type float

property tap_delays
Get the tap delays.

Returns The tap delays.

Return type np.ndarray

property tap_powers_dB
Get the tap powers (in dB).

Returns The tap powers (in dB).

Return type np.ndarray

property tap_powers_linear
Get the tap powers (in linear scale).

Returns The tap powers (in linear scale).

Return type np.ndarray

class pyphysim.channels.fading.**TdlImpulseResponse**(*tap_values: numpy.ndarray*, *channel_profile: pyphysim.channels.fading.TdlChannelProfile*)

Bases: object

Class that represents impulse response for a TdlChannel object.

This impulse response corresponds to the generated samples for one or more channel realization of the TdlChannel with the configured fading generator.

Parameters

- **tap_values** (*np.ndarray*) – The tap_values (not including zero padded taps) of a TDL channel generated for the non-zero taps. Dimension: *Num sparse taps x SHAPE x num_samples*. The value SHAPE here is the shape of the fading generator and corresponds to independent impulse responses. Often the shape of the used fading generator is None and thus the dimension of *tap_values* is just *Num sparse taps x num_samples*
- **channel_profile** (TdlChannelProfile) – The channel profile that was considering to generate this impulse response.

property Ts
Return the sampling interval of this impulse response.

If the impulse response is not discretized this returns None.

Returns The sampling interval.

Return type float

_get_samples_including_the_extra_zeros() → numpy.ndarray
Return the *samples* including the zeros for the zero taps.

Returns **samples_with_zeros** – The samples including the extra delays containing zeros.

Return type np.ndarray

property channel_profile
Return the channel profile.

Returns The channel profile.

Return type *TdlChannelProfile*

static concatenate_samples(*impulse_responses: List[TdlImpulseResponse]*) → *pyphysim.channels.fading.TdlImpulseResponse*
Concatenate multiple TdlImpulseResponse objects and return the new concatenated TdlImpulseResponse.

This concatenation is performed in the "samples" dimension.

Parameters **impulse_responses** (*list[TdlImpulseResponse]*) – A list of TdlImpulseResponse objects to be concatenated.

Returns The new concatenated TdlImpulseResponse.

Return type *TdlImpulseResponse*

get_freq_response(*fft_size: int*) → numpy.ndarray
Get the frequency response for this impulse response.

Parameters **fft_size** (*int*) – The size of the FFT to be applied.

Returns The frequency response. Dimension: *fft_size x num_samples* for SISO impulse response or *fft_size x num_rx x num_tx x num_samples* for MIMO impulse response.

Return type np.ndarray

property num_samples
Get the number of samples (different, "neighbor" impulse responses) stored here.

Returns The number of samples in the *TdlImpulseResponse* object.

Return type int

plot_frequency_response(*fft_size: int*) → None
Plot the frequency response.

Parameters **fft_size** (*int*) – The size of the FFT to be applied.

plot_impulse_response() → None
Plot the impulse response.

property tap_delays_sparse
Return the tap delays (which are multiples of the sampling interval).

Returns The tap delays.

Return type np.ndarray

property tap_indexes_sparse
Return the (sparse) tap indexes.

Returns

Return type The indexes of the non-zero taps.

property tap_values
Return the tap values (including zero padding) as a numpy array.

Returns The tap values (including zero padding).

Return type np.ndarray

property tap_values_sparse
Return the tap values (not including zero padding) as a numpy array.

Returns The tap values (not including possible zero padding).

Return type np.ndarray

class pyphysim.channels.fading.**TdlMimoChannel**(*fading_generator: Union[pyphysim.channels.fading_generators.JakesSampleGenerat pyphysim.channels.fading_generators.RayleighSampleGenerator]*, *channel_profile: Optional[pyphysim.channels.fading.TdlChannelProfile] = None*, *tap_powers_dB: Optional[numpy.ndarray] = None*, *tap_delays: Optional[numpy.ndarray] = None*, *Ts: Optional[float] = None*)

Bases: *pyphysim.channels.fading.TdlChannel*

Tapped Delay Line channel model, which corresponds to a multipath channel.

You can create a new TdlMimoChannel object either specifying the channel profile or specifying both the channel tap powers and delays.

Note that the TdlChannel class can already work with multiple antennas if provided *fading_generator* has a shape with two elements (number of receive antennas and number of transmit antennas). The TdlMimoChannel only adds a slight better interface over TdlChannel class for working with MIMO. This class is also useful to test MIMO transmission, with the added *num_tx_antennas* and *num_rx_antennas* properties.

Parameters

- **fading_generator** (*FadingGenerator*) – The instance of a fading generator in the *fading_generators* module. It should be a subclass of FadingSampleGenerator. The fading generator will be used to generate the channel samples. The shape of the fading_generator will be ignored and replaced by provided number of antennas.
- **channel_profile** (TdlChannelProfile) – The channel profile, which specifies the tap powers and delays.
- **tap_powers_dB** (*np.ndarray*) – The powers of each tap (in dB). Dimension: *L x 1* Note: The power of each tap will be a negative number (in dB).
- **tap_delays** (*np.ndarray*) – The delay of each tap (in seconds). Dimension: *L x 1*

pyphysim.channels.fading_generators module

class pyphysim.channels.fading_generators.**FadingSampleGenerator**(*shape: Optional[Union[int, Tuple[int, ...]]] = None*)

Bases: object

Base class for fading generators.

Parameters **shape** (*tuple[int] | int, optional*) – The shape of the sample generator. Each time *generate_more_samples(num_samples)* method is called it will generate samples with this shape as the first dimensions.

generate_more_samples(*num_samples: Optional[int] = None*) → None
Generate next samples.

When implementing this method in a subclass you must take the value of the self._shape attribute into account.

Parameters **num_samples** (*int, optional*) – Number of samples (with the provided shape) to generate. If not provided it will be assumed to be 1.

get_samples() → numpy.ndarray
Get the last generated sample.

Returns

Return type np.ndarray

get_similar_fading_generator() → Any
Get a similar fading generator with the same configuration, but that generates independent samples.

property shape
Get the shape of the sampling generator

This is the shape of the samples that will be generated (not including num_samples).

Returns

Return type tuple[int] | None

skip_samples_for_next_generation(*num_samples: int*) → None
Advance sample generation process by *num_samples* similarly to what would happen if you call *generate_more_samples(num_samples=num_samples)*, but without actually generating the samples.

Parameters **num_samples** (*int*) – How many samples to skip.

class pyphysim.channels.fading_generators.**JakesSampleGenerator**(*Fd: float = 100, Ts: float = 0.001, L: int = 8, shape: Optional[Union[int, Tuple[int, ...]]] = None, RS: Optional[numpy.random.mtrand.RandomSta = None*)

Bases: *pyphysim.channels.fading_generators.FadingSampleGenerator*

Class that generated fading samples according to the Jakes model given by

$$h(t) = \frac{1}{\sqrt{L}} \sum_{l=0}^{L-1} \exp\{j[2\pi f_D \cos(\phi_l)t + \psi_l]\}$$

Parameters

- **Fd** (*float*) – The Doppler frequency (in Hertz).
- **Ts** (*float*) – The sample interval (in seconds).
- **L** (*int*) – The number of rays for the Jakes model.
- **shape** (*int | tuple[int], optional*) – The shape of the sample generator. Each time the *generate_jakes_samples* method is called it will generate samples with this shape. If not provided, then 1 will be assumed. This could be used to generate MIMO channels. For instance, in order to generate channels samples for a MIMO scenario with 3 receive antennas and 2 transmit antennas use a shape of (3, 2).
- **RS** (*np.random.RandomState*) – The RandomState object used to generate the random values. If not provided, the global RandomState in numpy will be used.

See also:

generate_jakes_samples

property Fd
The Doppler frequency (in Hertz)

property L
The number of rays for the Jakes model

property Ts
The sample interval (in seconds)

_generate_time_samples(*num_samples: Optional[int] = None*) → numpy.ndarray
Generate the time samples that will be used internally in *generate_more_samples* method.

Parameters **num_samples** (*int, optional*) – Number of samples to be generated.

Returns The numpy array with the time samples. The shape of the generated time variable is "(1, A, num_samples)", where 'A' has as many '1's as the length of self._shape. Ex: If self._shape is None then the shape of the returned 't' variable is (1, num_samples). If self._shape is (2,3) then the shape of the returned 't' variable is (1, 1, 1, num_samples)

Return type np.ndarray

Notes

Each time *_generate_time_samples* is called it will update *_current_time* to reflect the advance of the time after generating the new samples.

_set_phi_and_psi_according_to_shape() → None
This will update the phi and psi attributes used to generate the jakes samples to reflect the current value of self._shape.

generate_more_samples(*num_samples: Optional[int] = None*) → None
Generate next samples.

Note that any subsequent call to this method continues from the point where the last call stopped. That is, if you generate 10 samples and then 15 more samples, you will get the same samples you would have got if you had generated 25 samples.

Parameters **num_samples** (*int, optional*) – Number of samples (with the provided shape) to generate. If not provided it will be assumed to be 1.

Notes

This method will update the self._current_time variable.

get_similar_fading_generator() → Any
Get a similar fading generator with the same configuration, but that generates independent samples.

Returns Another JakesSampleGenerator object with the same configuration of this object.

Return type *JakesSampleGenerator*

property shape
Get the shape of the sampling generator

This is the shape of the samples that will be generated (not including num_samples).

Returns

Return type tuple[int] | None

skip_samples_for_next_generation(*num_samples: int*) → None
Advance sample generation process by *num_samples* similarly to what would happen if you call *generate_more_samples(num_samples=num_samples)*, but without actually generating the samples.

This has the effect of advancing the internal time using by JakesSampleGenerator without generating any samples.

Parameters **num_samples** (*int*) – How many samples to skip.

class pyphysim.channels.fading_generators.**RayleighSampleGenerator**(*shape: Optional[Union[int, Tuple[int, ...]]] = None*)
Bases: *pyphysim.channels.fading_generators.FadingSampleGenerator*

Class that generates fading samples from a Raleigh distribution.

Parameters **shape** (*int | tuple[int] | None*) – The shape of the sample generator. Each time the *generate_jakes_samples* method is called it will generate samples with this shape. If not provided, then 1 will be assumed.

generate_more_samples(*num_samples: Optional[int] = None*) → None
Generate next samples.

Parameters **num_samples** (*int, optional*) – Number of samples (with the provided shape) to generate. If not provided it will be assumed to be 1.

get_similar_fading_generator() → Any
Get a similar fading generator with the same configuration, but that generates independent samples.

Returns Another RayleighSampleGenerator object with the same configuration of this object.

Return type *RayleighSampleGenerator*

skip_samples_for_next_generation(*num_samples: int*) → None
Advance sample generation process by *num_samples* similarly to what would happen if you call *generate_more_samples(num_samples=num_samples)*, but without actually generating the samples.

Since the samples generated by RayleighSampleGenerator are independent, calling this method has no effect.

Parameters **num_samples** (*int*) – How many samples to skip. This is ignored in the RayleighSampleGenerator. Since the different samples are uncorrelated then calling *skip_samples_for_next_generation* does not do anything.

pyphysim.channels.fading_generators.**generate_jakes_samples**(*Fd: float, Ts: float = 0.001, NSamples: int = 100, L: int = 8, shape: Optional[Tuple[int, ...]] = None, current_time: float = 0, phi_l: Optional[numpy.ndarray] = None, psi_l: Optional[numpy.ndarray] = None*) → Tuple[float, numpy.ndarray]

Generates channel samples according to the Jakes model.

This functions generates channel samples for a single tap according to the Jakes model given by

$$h(t) = \frac{1}{\sqrt{L}} \sum_{l=0}^{L-1} \exp\{j[2\pi f_D \cos(\phi_l)t + \psi_l]\} \tag{1.1}$$

Parameters

- **Fd** (*float*) – The Doppler frequency (in Hertz).
- **Ts** (*float*) – The sample interval (in seconds).
- **NSamples** (*int*) – The number of samples to generate.
- **L** (*int*) – The number of rays for the Jakes model.
- **shape** (*tuple[int]*) – The shape of the generated channel. This is used to generate MIMO channels. For instance, in order to generate channels samples for a MIMO scenario with 3 receive antennas and 2 transmit antennas use a shape of (3, 2).
- **current_time** (*float*) – The current start time
- **phi_l** (*np.ndarray*) – The "phi" part in Jakes model
- **psi_l** (*np.ndarray*) – The "psi" part in Jakes model

Returns

The **first element** in the returned tuple is the **new current time** (that should be used the next time this function is called to 'continue' the fading).

The **second** element in the returned tuple is the **generated channel**. If *shape* is None the the shape of the returned h is equal to (NSamples,). That is, h is a 1-dimensional numpy array. If *shape* was provided then the shape of h is the provided shape with an additional dimension for the time (the last dimension). For instance, if a *shape* of (3, 2) was provided then the shape of the returned h will be (3, 2, NSamples).

Return type (float, np.ndarray)

pyphysim.channels.multiuser module

Module containing multiuser channels.

The `MultiUserChannelMatrix` and `MultiUserChannelMatrixExtInt` classes implement the MIMO Interference Channel (MIMO-IC) model, where the first one does not include an external interference source while the last one includes it. The MIMO-IC model is shown in the Figure below.

Fig. 1: MIMO Interference Channel

class pyphysim.channels.multiuser.**MuChannel**(*N: Union[int, Tuple[int, int]], fading_generator: Optional[Union[pyphysim.channels.fading_generators.JakesSampleGen pyphysim.channels.fading_generators.RayleighSampleGenerator]] = None, channel_profile: Optional[pyphysim.channels.fading.TdlChannelProfile] = None, tap_powers_dB: Optional[numpy.ndarray] = None, tap_delays: Optional[numpy.ndarray] = None, Ts: Optional[float] = None*)

Bases: `object`

SISO multiuser channel.

Each transmitter sends data to its own receiver while interfering to other receivers.

Note that noise is NOT added.

Parameters

- **N** (*int | tuple[int,int]*) – The number of transmit/receive pairs.
- **fading_generator** (*T <= fading_generators.FadingSampleGenerator*) – The instance of a fading generator in the *fading_generators* module. It should be a subclass of FadingSampleGenerator. The fading generator will be used to generate the channel samples. However, since we have multiple links, the provided fading generator will actually be used to create similar (but independent) fading generators. If not provided then RayleighSampleGenerator will be used
- **channel_profile** (TdlChannelProfile) – The channel profile, which specifies the tap powers and delays.
- **tap_powers_dB** (*np.ndarray*) – The powers of each tap (in dB). Dimension: *L x 1* Note: The power of each tap will be a negative number (in dB).
- **tap_delays** (*np.ndarray*) – The delay of each tap (in seconds). Dimension: *L x 1*

Returns The created object.

Return type *MuChannel*

property channel_profile
Return the channel profile.

Returns The channel profile.

Return type *TdlChannelProfile*

corrupt_data(*signal: numpy.ndarray*) → numpy.ndarray
Corrupt data passed through the TDL channels of each link.

Note that noise is NOT added in *corrupt_data*.

Parameters **signal** (*np.ndarray*) – Signal to be transmitted through the channel. This should be a 2D numpy array (1D array if there is only one transmitter), where each row corresponds to the transmit data of one transmitter.

Returns Received signal at each receiver. Each row corresponds to one receiver.

Return type np.ndarray

corrupt_data_in_freq_domain(*signal: numpy.ndarray*, *fft_size: int*, *carrier_indexes: Union[numpy.ndarray, List[int], slice] = None*) → numpy.ndarray
Corrupt data passed through the TDL channels of each link, but in the frequency domain..

For each link, this is ROUGHLY equivalent to modulating *signal* with OFDM using *fft_size* subcarriers, transmitting through a regular TdlChannel, and then demodulating with OFDM to recover the received signal.

One important difference is that here the channel is considered constant during the transmission of *fft_size* elements in *signal*, and then it is varied by the equivalent of the variation for that number of elements. That is, the channel is block static.

Note that noise is NOT added in *corrupt_data*.

Parameters

- **signal** (*np.ndarray | list[np.ndarray]*) – Signal to be transmitted through the channel. This should be a 2D numpy array where each row corresponds to the transmit data of one transmitter. It can also be a list of numpy arrays or, if there is only one transmitter, a single 1D numpy array.
- **fft_size** (*int*) – The size of the Fourier transform to get the frequency response.
- **carrier_indexes** (*slice | np.ndarray | list[int]*) – The indexes of the subcarriers where signal is to be transmitted (all users will use the same indexes). If it is None assume all subcarriers will be used.

Returns Received signal at each receiver. Each row corresponds to one receiver.

Return type np.ndarray

get_last_impulse_response(*rx_idx: int*, *tx_idx: int*) → *pyphysim.channels.fading.TdlImpulseResponse*
Get the last generated impulse response.

A new impulse response is generated when the method *corrupt_data* is called. You can use the *get_last_impulse_response* method to get the impulse response used to corrupt the last data.

Parameters

- **rx_idx** (*int*) – The index of the receiver.

- **tx_idx** (*int*) – The index of the transmitter

Returns The impulse response of the channel that was used to corrupt the last data for the link from transmitter *tx_idx* to receiver *rx_idx*.

Return type *TdlImpulseResponse*

property num_rx_antennas
Get the number of receive antennas.

Returns The number of receive antennas.

Return type int

property num_taps
Get the number of taps in the profile.

Note that all links have the same channel profile.

Returns The number of taps in the channel (not including any zero padding).

Return type int

property num_taps_with_padding
Get the number of taps in the profile including zero-padding when the profile is discretized.

If the profile is not discretized an exception is raised.

Note that all links have the same channel profile.

Returns The number of taps in the channel (including any zero padding).

Return type int

property num_tx_antennas
Get the number of transmit antennas.

Returns The number of transmit antennas.

Return type np.ndarray

property pathloss_matrix
Get the matrix with the pathloss from each transmitter to each receiver.

Returns The pathloss matrix, if it was set, or None if there is no pathloss.

Return type np.ndarray

set_pathloss (*pathloss_matrix: numpy.ndarray*) → None
Set the path loss (IN LINEAR SCALE) from each transmitter to each receiver.

The path loss will be accounted when calling the corrupt_data method.

If you want to disable the path loss, set *pathloss_matrix* to None.

Parameters **pathloss_matrix** (*np.ndarray*) – A matrix with dimension "K x K", where K is the number of users, with the path loss (IN LINEAR SCALE) from each transmitter (columns) to each receiver (rows). If you want to disable the path loss then set it to None.

Notes

Note that path loss is a power relation, which means that the channel coefficients will be multiplied by the square root of elements in *pathloss_matrix*.

property switched_direction
Get the value of *switched_direction*.

Returns True if direction is switched and False otherwise.

Return type bool

class pyphysim.channels.multiuser.**MuMimoChannel**(*N: Union[int, Tuple[int, int]]*, *num_rx_antennas: int*, *num_tx_antennas: int*, *fading_generator: Optional[Union[*pyphysim.channels.fading_generators.JakesSamp pyphysim.channels.fading_generators.RayleighSampleGenerato *= None*, *channel_profile: Optional[*pyphysim.channels.fading.TdlChannelProfile*] = None*, *tap_powers_dB: Optional[numpy.ndarray] = None*, *tap_delays: Optional[numpy.ndarray] = None*, *Ts: Optional[float] = None*)

Bases: *pyphysim.channels.multiuser.MuChannel*

MIMO multiuser channel.

Each transmitter sends data to its own receiver while interfering to other receivers.

Note that noise is NOT added.

Parameters

- **N** (*int | tuple[int, int]*) – The number of transmit/receive pairs.
- **num_rx_antennas** (*int*) – Number of receive antennas of each user.
- **num_tx_antennas** (*int*) – Number of transmit antennas of each user.
- **fading_generator** (*T <= fading_generators.FadingSampleGenerator*) – The instance of a fading generator in the *fading_generators* module. It should be a subclass of FadingSampleGenerator. The fading generator will be used to generate the channel samples. However, since we have multiple links, the provided fading generator will actually be used to create similar (but independent) fading generators. If not provided then RayleighSampleGenerator will be used
- **channel_profile** (TdlChannelProfile) – The channel profile, which specifies the tap powers and delays.
- **tap_powers_dB** (*np.ndarray*) – The powers of each tap (in dB). Dimension: *L x 1* Note: The power of each tap will be a negative number (in dB).
- **tap_delays** (*np.ndarray*) – The delay of each tap (in seconds). Dimension: *L x 1*

class pyphysim.channels.multiuser.**MultiUserChannelMatrix**
Bases: object

Stores the (fast fading) channel matrix of a multi-user scenario. The path-loss from each transmitter to each receiver is also be accounted if the set_pathloss is called to set the path-loss matrix.

This channel matrix can be seem as an concatenation of blocks (of non-uniform size) where each block is a channel from one transmitter to one receiver and the block size is equal to the number of receive antennas of the receiver times the number of transmit antennas of the transmitter.

For instance, in a 3-users scenario the block (1,0) corresponds to the channel between the transmit antennas of user 0 and the receive antennas of user 1 (indexing staring at zero). If the number of receive antennas and transmit antennas of the three users are [2, 4, 6] and [2, 3, 5], respectively, then the block (1,0) would have a dimension of 4x2. Likewise, the channel matrix would look similar to the block structure below.

2 x 2	2 x 3	2 x 5
4 x 2	4 x 3	4 x 5
6 x 2	6 x 3	6 x 5

It is possible to initialize the channel matrix randomly by calling the *randomize* method, or from a given matrix by calling the *init_from_channel_matrix* method.

In order to get the channel matrix of a specific user *k* to another user *l*, call the *get_Hkl* method.

property H
Get method for the H property.

Returns The channel from all transmitters to all receivers. This is a numpy array of numpy arrays.

Return type np.ndarray

property K
Get method for the K property.

Returns The number of users (transmit-receive pairs).

Return type int

property Nr
Get method for the Nr property.

Returns The number of receive antennas of all users.

Return type np.ndarray

property Nt
Get method for the Nt property.

Returns The number of transmit antennas of all users.

Return type np.ndarray

property W
Post processing filters (a list of 2D numpy arrays) for each user.

Returns The Post processing filters for each user.

Return type list[np.ndarray]

_calc_Bkl_cov_matrix_all_l(*F_all_users: numpy.ndarray*, *k: int*, *N0_or_Rek: NumberOrArray = 0.0*) → numpy.ndarray
Calculates the interference-plus-noise covariance matrix for all streams at receiver k according to equation (28) in [Cadambe2008]_.

The interference-plus-noise covariance matrix for stream l of user k is given by Equation (28) in [Cadambe2008], which is reproduced below

$$^{[kl]} = \sum_{j=1}^{K} \frac{P^{[j]}}{d^{[j]}} \sum_{d=1}^{d^{[j]}} {}^{[kj]}_{\star l}{}^{[j]}_{\star l}{}^{[j]\dagger}{}^{[kj]\dagger} - \frac{P^{[k]}}{d^{[k]}} {}^{[kk]}_{\star l}{}^{[k]}_{\star l}{}^{[k]\dagger}{}^{[kk]\dagger} + {}_{N^{[k]}}$$

where $P^{[k]}$ is the transmit power of transmitter k, $d^{[k]}$ is the number of degrees of freedom of user k, $^{[kj]}$ is the channel between transmitter j and receiver k, $_{\star l}$ is the l-th column of the precoder of user k and $_{N^k}$ is an identity matrix with size equal to the number of receive antennas of receiver k.

Parameters

- **F_all_users** (*list[np.ndarray] | np.ndarray*) – The precoder of all users (already taking into account the transmit power). This can be a list of numpy arrays or a 1D numpy array of numpy arrays.
- **k** (*int*) – Index of the desired user.
- **N0_or_Rek** (*float | np.ndarray*) – If this is a 2D numpy array, it is interpreted as the covariance matrix of any external interference plus noise. If this is a number, it is interpreted as the noise power, in which case the covariance matrix will be an identity matrix times this noise power.

Returns **Bkl** – Covariance matrix of all streams of user k. Each element of the returned 1D numpy array is a 2D numpy complex array corresponding to the covariance matrix of one stream of user k.

Return type np.ndarray

Notes

To be simple, a function that returns the covariance matrix of only a single stream "l" of the desired user "k" could be implemented, but in the order to calculate the max SINR algorithm we need the covariance matrix of all streams and returning them in single function as is done here allows us to calculate the first part in equation (28) of [Cadambe2008] only once, since it is the same for all streams.

_calc_Bkl_cov_matrix_first_part(*F_all_users: numpy.ndarray*, *k: int*, *N0_or_Rek: NumberOrArray = 0.0*) → numpy.ndarray

Calculates the first part in the equation of the Blk covariance matrix in equation (28) of [Cadambe2008].

The first part is given by

$\sum_{j=1}^{K} \frac{P^{[j]}}{d^{[j]}} \sum_{d=1}^{d^{[j]}} {}^{[kj]}_{\star d} {}^{[j]}_{\star d} [j]\dagger [kj]\dagger + {}_{Nk}$

Note that it only depends on the value of k.

Parameters

- **F_all_users** (*list[np.ndarray] | np.ndarray*) – The precoder of all users (already taking into account the transmit power). It can be a list of numpy arrays or a numpy array of numpy arrays.
- **k** (*int*) – Index of the desired user.
- **N0_or_Rek** (*float | np.ndarray*) – If this is a 2D numpy array, it is interpreted as the covariance matrix of any external interference plus noise. If this is a number, it is interpreted as the noise power, in which case the covariance matrix will be an identity matrix times this noise power.

Returns **first_part**

Return type np.ndarray

_calc_Bkl_cov_matrix_second_part(*Fk: numpy.ndarray*, *k: int*, *l: int*) → numpy.ndarray

Calculates the second part in the equation of the Blk covariance matrix in equation (28) of [Cadambe2008] (note that it does not include the identity matrix).

The second part is given by

$\frac{P^{[k]}}{d^{[k]}} {}^{[kk]}_{\star l} {}^{[k]}_{\star l} [k]\dagger [kk]\dagger$

Parameters

- **Fk** (*np.ndarray*) – The precoder of the desired user.
- **k** (*int*) – Index of the desired user.
- **l** (*int*) – Index of the desired stream.

Returns **second_part** – Second part in equation (28) of [Cadambe2008].

Return type np.ndarray

_calc_JP_Bkl_cov_matrix_all_l(*F_all_users: numpy.ndarray*, *k: int*, *N0_or_Rek: NumberOrArray = 0.0*) → numpy.ndarray

Calculates the interference-plus-noise covariance matrix for all streams at receiver k according to equation (28) in [Cadambe2008]_.

The interference-plus-noise covariance matrix for stream l of user k is given by Equation (28) in [Cadambe2008], which is reproduced below

$${}^{[kl]} = \sum_{j=1}^{K} \frac{P^{[j]}}{d^{[j]}} \sum_{d=1}^{d^{[j]}} {}^{[kj]}_{\star l} {}^{[j]}_{\star l} [j]\dagger [kj]\dagger - \frac{P^{[k]}}{d^{[k]}} {}^{[kk]}_{\star l} {}^{[k]}_{\star l} [k]\dagger [kk]\dagger + {}_{N^{[k]}}$$

where $P^{[k]}$ is the transmit power of transmitter k, $d^{[k]}$ is the number of degrees of freedom of user k, ${}^{[kj]}$ is the channel between transmitter j and receiver k, ${}_{\star l}$ is the l-th column of the precoder of user k and ${}_{N^k}$ is an identity matrix with size equal to the number of receive antennas of receiver k.

Parameters

- **F_all_users** (*list[np.ndarray] | np.ndarray*) – The precoder of all users (already taking into account the transmit power). This can be either a 1D numpy array of numpy arrays or a list of numpy arrays.
- **k** (*int*) – Index of the desired user.
- **N0_or_Rek** (*float | np.ndarray*) – If this is a 2D numpy array, it is interpreted as the covariance matrix of any external interference plus noise. If this is a number, it is interpreted as the noise power, in which case the covariance matrix will be an identity matrix times this noise power.

Returns **Bkl** – Covariance matrix of all streams of user k. Each element of the returned 1D numpy array is a 2D numpy complex array corresponding to the covariance matrix of one stream of user k.

Return type np.ndarray

Notes

To be simple, a function that returns the covariance matrix of only a single stream "l" of the desired user "k" could be implemented, but in the order to calculate the max SINR algorithm we need the covariance matrix of all streams and returning them in single function as is done here allows us to calculate the first part in equation (28) of [Cadambe2008] only once, since it is the same for all streams.

_calc_JP_Bkl_cov_matrix_first_part(*F_all_users: numpy.ndarray*, *k: int*, *noise_power: float = 0.0*) → numpy.ndarray

Calculates the first part in the equation of the Blk covariance matrix in equation (28) of [Cadambe2008] when joint process is employed.

The first part is given by

$\sum_{j=1}^{K} \frac{P^{[j]}}{d^{[j]}} \sum_{d=1}^{d^{[j]}} {}^{[kj]}_{\star d}[j]{}^{[j]\dagger}_{\star d}[kj]^{\dagger} + N k$

Note that it only depends on the value of k.

Parameters

- **F_all_users** (*list[np.ndarray] | np.ndarray*) – The precoder of all users (already taking into account the transmit power). It can be a list of numpy arrays or a numpy array of numpy arrays.
- **k** (*int*) – Index of the desired user.
- **noise_power** (*float | None, optional*) – The noise power.

Returns

Return type np.ndarray

_calc_JP_Bkl_cov_matrix_first_part_impl(*Hk: numpy.ndarray, F_all_users: numpy.ndarray, Rek: NumberOrArray*) → numpy.ndarray

Common implementation of the _calc_JP_Bkl_cov_matrix_first_part.

Parameters

- **Hk** (*np.ndarray*) – The channel from all transmitters (not including external interference source, if any) to receiver k.
- **F_all_users** (*list[np.ndarray]*) – The precoder of all users (already taking into account the transmit power).
- **Rek** (*np.ndarray | float*) – Covariance matrix of the external interference (if there is any) plus noise.

Returns The *first_part* for the Bkl matrix computation.

Return type np.ndarray

_calc_JP_Bkl_cov_matrix_second_part(*Fk: numpy.ndarray, k: int, l: int*) → numpy.ndarray

Calculates the second part in the equation of the Blk covariance matrix in equation (28) of [Cadambe2008] (note that it does not include the identity matrix).

The second part is given by

$\frac{P^{[k]}}{d^{[k]}} {}^{[kk]}_{\star l}[k]{}^{[k]\dagger}_{\star l}[kk]^{\dagger}$

Parameters

- **Fk** (*np.ndarray*) – The precoder of the desired user.
- **k** (*int*) – Index of the desired user.
- **l** (*int*) – Index of the desired stream.

Returns **second_part** – Second part in equation (28) of [Cadambe2008].

Return type np.ndarray.

static _calc_JP_Bkl_cov_matrix_second_part_impl(*Hk: numpy.ndarray, Fk: numpy.ndarray, l: int*) → numpy.ndarray

Common implementation of the _calc_JP_Bkl_cov_matrix_second_part method.

Parameters

- **Hk** (*np.ndarray*) –
- **Fk** (*np.ndarray*) –
- **l** (*int*) –

Returns

Return type np.ndarray

_calc_JP_Q_impl(*k: int*, *F_all_users: numpy.ndarray*) → numpy.ndarray

Calculates the interference covariance matrix (without any noise) at the k-th receiver with a joint processing scheme.

See the documentation of the calc_JP_Q method.

Parameters

- **k** (*int*) – The user index.
- **F_all_users** (*list[np.ndarray] | np.ndarray*) – The precoders of all users. It can be a list of numpy arrays or a numpy array of numpy arrays.

Returns The interference covariance matrix (without any noise).

Return type np.ndarray

_calc_JP_SINR_k(*k: int*, *Fk: numpy.ndarray*, *Uk: numpy.ndarray*, *Bkl_all_l: numpy.ndarray*) → numpy.ndarray

Calculates the SINR of all streams of user 'k'.

Parameters

- **k** (*int*) – Index of the desired user.
- **Fk** (*np.ndarray*) – The precoder of user k.
- **Uk** (*np.ndarray*) – The receive filter of user k (before applying the conjugate transpose).
- **Bkl_all_l** (*list[np.ndarray] | np.ndarray*) – A sequence (1D numpy array, a list, etc) of 2D numpy arrays corresponding to the Bkl matrices for all 'l's.

Returns **SINR_k** – The SINR for the different streams of user k.

Return type np.ndarray

static _calc_JP_SINR_k_impl(*Hk: numpy.ndarray*, *Fk: numpy.ndarray*, *Uk: numpy.ndarray*, *Bkl_all_l: numpy.ndarray*) → numpy.ndarray

Implementation of the *_calc_JP_SINR_k()* method.

Parameters

- **Hk** (*np.ndarray*) – Channel from all transmitters to receiver k.
- **Fk** (*np.ndarray*) – The precoder of user k.
- **Uk** (*np.ndarray*) – The receive filter of user k (before applying the conjugate transpose).
- **Bkl_all_l** (*list[np.ndarray]*) – A sequence (1D numpy array, a list, etc) of 2D numpy arrays corresponding to the Bkl matrices for all 'l's.

Returns **SINR_k** – The SINR for the different streams of user k.

Return type np.ndarray

Notes

The implementation of the _calc_JP_SINR_k method is almost the same for the MultiuserChannelMatrix and MultiuserChannelMatrixExtint class, except for the *Hk* argument. Therefore, the common code was put here and in each class the `_calc_JP_SINR_k()` is implemented as simply getting the correct *Hk* argument and then calling `_calc_JP_SINR_k_impl()`.

_calc_Q_impl(*k: int*, *F_all_users: numpy.ndarray*) → numpy.ndarray

Calculates the interference covariance matrix (without any noise) at the k-th receiver.

See the documentation of the calc_Q method.

Parameters

- **k** (*int*) – Index of the desired receiver.
- **F_all_users** (*np.ndarray*) – The precoder of all users (already taking into account the transmit power). This should be a 1D numpy array of 2D numpy arrays.

Returns

Return type np.ndarray

_calc_SINR_k(*k: int*, *Fk: numpy.ndarray*, *Uk: numpy.ndarray*, *Bkl_all_l: numpy.ndarray*) → numpy.ndarray

Calculates the SINR of all streams of user 'k'.

Parameters

- **k** (*int*) – Index of the desired user.
- **Fk** (*np.ndarray*) – The precoder of user k.
- **Uk** (*np.ndarray*) – The receive filter of user k (before applying the conjugate transpose).
- **Bkl_all_l** (*list[np.ndarray] | np.ndarray*) – A sequence (1D numpy array, a list, etc) of 2D numpy arrays corresponding to the Bkl matrices for all 'l's.

Returns **SINR_k** – The SINR for the different streams of user k.

Return type np.ndarray

static _from_small_matrix_to_big_matrix(*small_matrix: numpy.ndarray*, *Nr: numpy.ndarray*, *Nt: numpy.ndarray*, *Kr: int*, *Kt: Optional[int] = None*) → numpy.ndarray

Convert from a small matrix to a big matrix by repeating elements according to the number of receive and transmit antennas.

Parameters

- **small_matrix** (*np.ndarray*) – Any 2D numpy array
- **Nr** (*np.ndarray*) – Number of antennas at each receiver. This should be a 1D numpy array.
- **Nt** (*np.ndarray*) – Number of antennas at each transmitter. This should be a 1D numpy array.
- **Kr** (*int*) – Number of receivers to consider.
- **Kt** (*int, optional*) – Number of transmitters to consider. It not provided the value of Kr will be used.

Returns **big_matrix** – The converted matrix. This is a 2D numpy array

Return type np.ndarray

Notes

Since a 'user' is a transmit/receive pair then the small_matrix will be a square matrix and *Kr* must be equal to *Kt*. However, in the *MultiUserChannelMatrixExtInt* class we will only have the 'transmitter part' for the external interference sources. That means that small_matrix will have more columns then rows and *Kt* will be greater then *Kr*.

Examples

```
>>> K = 3
>>> Nr = np.array([2, 4, 6])
>>> Nt = np.array([2, 3, 5])
>>> small_matrix = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> MultiUserChannelMatrix._from_small_matrix_to_big_matrix(
↪small_matrix, Nr, Nt, K)
array([[1, 1, 2, 2, 2, 3, 3, 3, 3, 3],
       [1, 1, 2, 2, 2, 3, 3, 3, 3, 3],
       [4, 4, 5, 5, 5, 6, 6, 6, 6, 6],
       [4, 4, 5, 5, 5, 6, 6, 6, 6, 6],
       [4, 4, 5, 5, 5, 6, 6, 6, 6, 6],
       [4, 4, 5, 5, 5, 6, 6, 6, 6, 6],
       [7, 7, 8, 8, 8, 9, 9, 9, 9, 9],
       [7, 7, 8, 8, 8, 9, 9, 9, 9, 9],
       [7, 7, 8, 8, 8, 9, 9, 9, 9, 9],
       [7, 7, 8, 8, 8, 9, 9, 9, 9, 9],
       [7, 7, 8, 8, 8, 9, 9, 9, 9, 9],
       [7, 7, 8, 8, 8, 9, 9, 9, 9, 9]])
```

property big_H
Get method for the big_H property.

Returns The channel from all transmitters to all receivers as a single big matrix (numpy complex array)

Return type np.ndarray

property big_W
Post processing filters (a block diagonal matrix) for each user.

Returns The big block diagonal matrix with the post processing filters for each user.

Return type np.ndarray

calc_JP_Q(*k: int*, *F_all_users: numpy.ndarray*) → numpy.ndarray
Calculates the interference plus noise covariance matrix at the k-th receiver with a joint processing scheme.

The interference covariance matrix at the k-th receiver, k, is given by

$$k = \sum_{j=1, j \neq k}^{K} \frac{P_j}{Ns_j} {}_k j_j^H {}_k^H$$

where P_j is the transmit power of transmitter j, and Ns_j is the number of streams for user j.

Parameters

- **k** (*int*) – Index of the desired receiver.
- **F_all_users** (*np.ndarray | list[np.ndarray]*) – The precoder of all users (already taking into account the transmit power).

Returns **Qk** – The interference covariance matrix at receiver k.

Return type np.ndarray

calc_JP_SINR(*F: numpy.ndarray*, *U: numpy.ndarray*) → numpy.ndarray
Calculates the SINR values (in linear scale) of all streams of all users with the current IA solution.

The noise variance used will be the value of the noise_var property.

Parameters

- **F** (*np.ndarray*) – The precoders of all users. This should be a 1D numpy array of 2D numpy arrays.
- **U** (*np.ndarray*) – The receive filters of all users. This should be a 1D numpy array of 2D numpy arrays.

Returns **SINRs** – The SINR (in linear scale) of all streams of all users. This is a 1D numpy array of 1D numpy arrays (of floats).

Return type np.ndarray

calc_Q(*k: int*, *F_all_users: numpy.ndarray*) → numpy.ndarray
Calculates the interference plus noise covariance matrix at the k-th receiver.

The interference covariance matrix at the k-th receiver, k, is given by

$k = \sum_{j=1, j \neq k}^{K} \frac{P_j}{Ns_j} {}_{kj}{}^{j}{}_{j}^{H} {}_{kj}^{H}$

where P_j is the transmit power of transmitter j, and Ns_j is the number of streams for user j.

Parameters

- **k** (*int*) – Index of the desired receiver.
- **F_all_users** (*np.ndarray*) – The precoder of all users (already taking into account the transmit power). This should be a 1D numpy array of 2D numpy arrays.

Returns **Qk** – The interference covariance matrix at receiver k (a 2D numpy complex array).

Return type np.ndarray

calc_SINR(*F: numpy.ndarray*, *U: numpy.ndarray*) → numpy.ndarray
Calculates the SINR values (in linear scale) of all streams of all users with the current IA solution.

The noise variance used will be the value of the noise_var property.

Parameters

- **F** (*np.ndarray*) – The precoders of all users. This should be a 1D numpy array of 2D numpy arrays.
- **U** (*np.ndarray*) – The receive filters of all users. This should be a 1D numpy array of 2D numpy arrays.

Returns **SINRs** – The SINR (in linear scale) of all streams of all users. This is a 1D numpy array of 1D numpy arrays (of floats)

Return type np.ndarray

corrupt_concatenated_data(*data: numpy.ndarray*) → numpy.ndarray
Corrupt data passed through the channel.

If self.noise_var is set to some scalar number then white noise will also be added.

Parameters **data** (*np.ndarray*) – A bi-dimensional numpy array with the concatenated data of all transmitters. The dimension of data is sum(self.Nt) x NSymb. That is, the number of rows corresponds to the sum of the number of transmit antennas of all users and the number of columns correspond to the number of transmitted symbols.

Returns A bi-dimension numpy array where the number of rows corresponds to the sum of the number of receive antennas of all users and the number of columns correspond to the number of transmitted symbols.

Return type np.ndarray

corrupt_data (*data: numpy.ndarray*) → numpy.ndarray

Corrupt data passed through the channel.

If the noise_var is supplied then an white noise will also be added.

Parameters **data** (*np.ndarray*) – An array of numpy matrices with the data of the multiple users. The k-th element in *data* is a numpy array with dimension Nt_k x NSymbs, where Nt_k is the number of transmit antennas of the k-th user and NSymbs is the number of transmitted symbols.

Returns A numpy array where each element contains the received data (a 2D numpy array) of a user.

Return type np.ndarray

get_Hk (*k: int*) → numpy.ndarray

Get the channel from all transmitters to receiver *k*.

Parameters **k** (*int*) – Receiving user.

Returns **channel_k** – Channel from all transmitters to receiver *k*. This is a 2D numpy array.

Return type np.ndarray

See also:

get_Hkl()

Examples

```
>>> multiH = MultiUserChannelMatrix()
>>> H = np.reshape(np.r_[0:16], [4,4])
>>> Nt = np.array([2, 2])
>>> Nr = np.array([2, 2])
>>> multiH.init_from_channel_matrix(H, Nr, Nt, 2)
>>> print(multiH.big_H)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
>>> print(multiH.get_Hk(0))
[[0 1 2 3]
 [4 5 6 7]]
>>> print(multiH.get_Hk(1))
[[ 8  9 10 11]
 [12 13 14 15]]
```

get_Hkl (*k: int*, *l: int*) → numpy.ndarray

Get the channel matrix from user *l* to user *k*.

Parameters

- **l** (*int*) – Transmitting user.
- **k** (*int*) – Receiving user.

Returns **channel** – Channel from transmitter *l* to receiver *k*. This is a 2D numpy array.

Return type np.ndarray

See also:

get_Hk()

Examples

```
>>> multiH = MultiUserChannelMatrix()
>>> H = np.reshape(np.r_[0:16], [4,4])
>>> Nt = np.array([2, 2])
>>> Nr = np.array([2, 2])
>>> multiH.init_from_channel_matrix(H, Nr, Nt, 2)
>>> print(multiH.big_H)
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
>>> print(multiH.get_Hkl(0, 0))
[[0 1]
 [4 5]]
>>> print(multiH.get_Hkl(1, 0))
[[ 8  9]
 [12 13]]
```

init_from_channel_matrix(*channel_matrix: numpy.ndarray*, *Nr: Union[numpy.ndarray, int]*, *Nt: Union[numpy.ndarray, int]*, *K: int*) → None

Initializes the multiuser channel matrix from the given *channel_matrix*.

Parameters

- **channel_matrix** (*np.ndarray*) – A matrix concatenating the channel of all users (from each transmitter to each receiver). This is a 2D numpy array.
- **Nr** (*int | np.ndarray*) – Number of antennas at each receiver.
- **Nt** (*int | np.ndarray*) – Number of antennas at each transmitter.
- **K** (*int*) – Number of transmit/receive pairs.

Raises **ValueError** – If the arguments are invalid.

property last_noise

Get method for the last_noise property.

Returns The last AWGN noise array added to corrupt the data.

Return type None | np.ndarray

property noise_var

Get method for the noise_var property.

Returns The noise variance, if noise is being added in "corrupt_*data" methods.

Return type None | float

property pathloss
Get method for the pathloss property.

Returns The pathloss matrix (if one was set).

Return type None | np.ndarray

randomize(*Nr: Union[numpy.ndarray, int]*, *Nt: Union[numpy.ndarray, int]*, *K: int*) → None
Generates a random channel matrix for all users.

Parameters

- **Nr** (*int | np.ndarray*) – Number of receive antennas of each user. If an integer is specified, all users will have that number of receive antennas.
- **Nt** (*int | np.ndarray*) – Number of transmit antennas of each user. If an integer is specified, all users will have that number of receive antennas.
- **K** (*int*) – Number of users.

re_seed() → None
Re-seed the channel and noise RandomState objects randomly.

If you want to specify the seed for each of them call the *set_channel_seed* and *set_noise_seed* methods and pass the desired seed for each of them.

set_channel_seed(*seed: Optional[Union[int, List[int], numpy.ndarray]] = None*) → None
Set the seed of the RandomState object used to generate the random elements of the channel (when self.randomize is called).

Parameters **seed** (*None | int | array_like*) – Random seed initializing the pseudo-random number generator. See np.random.RandomState help for more info.

set_noise_seed(*seed: Optional[Union[int, List[int], numpy.ndarray]] = None*) → None
Set the seed of the RandomState object used to generate the random noise elements (when the corrupt data function is called).

Parameters **seed** (*None | int | array_like*) – Random seed initializing the pseudo-random number generator. See np.random.RandomState help for more info.

set_pathloss(*pathloss_matrix: Optional[numpy.ndarray] = None*) → None
Set the path loss (IN LINEAR SCALE) from each transmitter to each receiver.

The path loss will be accounted when calling the get_Hkl, get_Hk, the corrupt_concatenated_data and the corrupt_data methods.

If you want to disable the path loss, set *pathloss_matrix* to None.

Parameters **pathloss_matrix** (*np.ndarray*) – A matrix with dimension "K x K", where K is the number of users, with the path loss (IN LINEAR SCALE) from each transmitter (columns) to each receiver (rows). If you want to disable the path loss then set it to None.

Notes

Note that path loss is a power relation, which means that the channel coefficients will be multiplied by the square root of elements in *pathloss_matrix*.

set_post_filter (*filters: numpy.ndarray*) → None
Set the post-processing filters.

The post-processing filters will be applied to the data after if has been corrupted by the channel in either the *corrupt_data* or the *corrupt_concatenated_data* methods.

Parameters **filters** (*list[np.ndarray] | np.ndarray*) – The post processing filters of each user. This should be a list of 2D np arrays or a 1D np array of 2D np arrays.

class pyphysim.channels.multiuser.**MultiUserChannelMatrixExtInt**
Bases: *pyphysim.channels.multiuser.MultiUserChannelMatrix*

Very similar to the MultiUserChannelMatrix class, but the MultiUserChannelMatrixExtInt also includes the effect of an external interference.

This channel matrix can be seem as an concatenation of blocks (of non-uniform size) where each block is a channel from one transmitter to one receiver and the block size is equal to the number of receive antennas of the receiver times the number of transmit antennas of the transmitter. The difference compared with MultiUserChannelMatrix is that in the MultiUserChannelMatrixExtInt class the interference user counts as one more user, but with zero receive antennas.

For instance, in a 3-users scenario the block (1,0) corresponds to the channel between the transmit antennas of user 0 and the receive antennas of user 1 (indexing staring at zero). If the number of receive antennas and transmit antennas of the three users are [2, 4, 6] and [2, 3, 5], respectively, then the block (1,0) would have a dimension of 4x2. The external interference will count as one more block where the number of columns of this block corresponds to the rank of the external interference. If the external interference has a rank 2 then the complete channel matrix would look similar to the block structure below.

2 x 2	2 x 3	2 x 5	2 x 2
4 x 2	4 x 3	4 x 5	4 x 2
6 x 2	6 x 3	6 x 5	6 x 2

The methods from the MultiUserChannelMatrix class that makes sense were reimplemented here to include information regarding the external interference.

property H
Get method for the H property.

property H_no_ext_int
Get method for the H_no_ext_int property.

property K
Get method for the K property.

property Nr
Get method for the Nr property.

property Nt
Get method for the Nt property.

_calc_JP_Bkl_cov_matrix_first_part (*F_all_users: numpy.ndarray*, *k: int*, *Rek: NumberOrArray*) → numpy.ndarray
Calculates the first part in the equation of the Blk covariance matrix in equation (28) of [Cadambe2008] when joint process is employed.

The first part is given by

$\sum_{j=1}^{K} \frac{P^{[j]}}{d^{[j]}} \sum_{d=1}^{d^{[j]}} {}^{[kj]}_{\star d}[j]{}^{[j]\dagger}_{\star d}[kj]^{\dagger} + N_k$

Note that it only depends on the value of k.

Parameters

- **F_all_users** (*list[np.ndarray] | np.ndarray*) – The precoder of all users (already taking into account the transmit power). This can be either a 1D numpy array of numpy arrays or a list of numpy arrays.
- **k** (*int*) – Index of the desired user.
- **Rek** (*np.ndarray | float*) – Covariance matrix of the external interference plus noise.

Returns The first part in the equation of the Blk covariance matrix.

Return type np.ndarray

_calc_JP_Bkl_cov_matrix_second_part(*Fk: numpy.ndarray*, *k: int*, *l: int*) → numpy.ndarray

Calculates the second part in the equation of the Blk covariance matrix in equation (28) of [Cadambe2008] (note that it does not include the identity matrix).

The second part is given by

$\frac{P^{[k]}}{d^{[k]}} {}^{[kk]}_{\star l}[k]{}^{[k]\dagger}_{\star l}[kk]^{\dagger}$

Parameters

- **Fk** (*np.ndarray*) – The precoder of the desired user.
- **k** (*int*) – Index of the desired user.
- **l** (*int*) – Index of the desired stream.

Returns **second_part** – Second part in equation (28) of [Cadambe2008].

Return type np.ndarray

_calc_JP_Q(*k: int*, *F_all_users: numpy.ndarray*) → numpy.ndarray

Calculates the interference covariance matrix at the k-th receiver with a joint processing scheme (not including the covariance matrix of the external interference plus noise)

Parameters

- **k** (*int*) – Index of the desired receiver.
- **F_all_users** (*np.ndarray*) – The precoder of all users (already taking into account the transmit power). This is a 1D numpy array of 2D numpy array.

See also:

calc_JP_Q()

_calc_JP_SINR_k(*k: int*, *Fk: numpy.ndarray*, *Uk: numpy.ndarray*, *Bkl_all_l: numpy.ndarray*) → numpy.ndarray

Calculates the SINR of all streams of user 'k'.

Parameters

- **Fk** (*np.ndarray*) – The precoder of user k.
- **Uk** (*np.ndarray*) – The receive filter of user k (before applying the conjugate transpose).

- **k** (*int*) – Index of the desired user.
- **Bkl_all_l** (*list[np.ndarray] | np.ndarray*) – A sequence (1D numpy array, a list, etc) of 2D numpy arrays corresponding to the Bkl matrices for all 'l's.

Returns The SINR for the different streams of user k.

Return type np.ndarray

static _prepare_input_parans(*Nr: numpy.ndarray*, *Nt: numpy.ndarray*, *K: int*, *NtE: Iterable[int]*) → Tuple[numpy.ndarray, numpy.ndarray, int, int, numpy.ndarray]

Helper method used in the init_from_channel_matrix and randomize method definitions.

Parameters

- **Nr** (*np.ndarray*) – Number of antennas at each receiver.
- **Nt** (*np.ndarray*) – Number of antennas at each transmitter.
- **K** (*int*) – Number of transmit/receive pairs.
- **NtE** (*int | list[int] | np.ndarray*) – Number of transmit antennas of the external interference source(s). If *NtE* is an iterable, the number of external interference sources will be the len(NtE).

Returns **output** – The tuple (full_Nr, full_Nt, full_K, extIntK, extIntNt).

Return type tuple

property big_H_no_ext_int

Get method for the big_H_no_est_int property.

big_H_no_est_int is similar to big_H, but does not include the last column(s) corresponding to the external interference channel.

calc_JP_Q(*k: int*, *F_all_users: numpy.ndarray*, *pe: float = 1.0*) → numpy.ndarray

Calculates the interference covariance matrix at the k-th receiver with a joint processing scheme.

The interference covariance matrix at the k-th receiver, k, is given by

$k = \sum_{j=1,j\neq k}^{K} \frac{P_j}{Ns_j} {}_k{}^j H_j^H H_k$

where P_j is the transmit power of transmitter j, and Ns_j is the number of streams for user j.

Parameters

- **k** (*int*) – Index of the desired receiver.
- **F_all_users** (*list[np.ndarray] | np.ndarray*) – The precoder of all users (already taking into account the transmit power). This is a 1D numpy array of 2D numpy array.
- **pe** (*float*) – The power of the external interference source(s).

Returns **Qk** – The interference covariance matrix at receiver k.

Return type np.ndarray

calc_JP_SINR(*F: numpy.ndarray*, *U: numpy.ndarray*, *pe: float = 1.0*) → numpy.ndarray

Calculates the SINR values (in linear scale) of all streams of all users with the current IA solution.

The noise variance used will be the value of the noise_var property.

Parameters

- **F** (*np.ndarray*) – The precoders of all users. This is a 1D numpy array of 2D numpy arrays.
- **U** (*np.ndarray*) – The receive filters of all users. This is a 1D numpy array of 2D numpy arrays.
- **pe** (*float*) – The external interference power.

Returns The SINR (in linear scale) of all streams of all users. This is a 1D numpy array of 1D numpy arrays (of floats).

Return type np.ndarray

calc_Q(*k: int*, *F_all_users: numpy.ndarray*, *pe: float = 1.0*) → numpy.ndarray

Calculates the interference covariance matrix at the k-th receiver.

The interference covariance matrix at the k-th receiver, k, is given by

$$k = \sum_{j=1, j \neq k}^{K} \frac{P_j}{Ns_j} {}_{kj}{}^{j}{}^{H}_{j}{}^{H}_{kj}$$

where P_j is the transmit power of transmitter j, and Ns_j is the number of streams for user j.

Parameters

- **k** (*int*) – Index of the desired receiver.
- **F_all_users** (*list[np.ndarray] | np.ndarray*) – The precoder of all users (already taking into account the transmit power). This should be either a list of numpy 2D arrays or a 1D numpy array of 2D numpy arrays.
- **pe** (*float*) – The power of the external interference source(s).

Returns **Qk** – The interference covariance matrix at receiver k.

Return type np.ndarray

calc_SINR(*F: numpy.ndarray*, *U: numpy.ndarray*, *pe: float = 1.0*) → numpy.ndarray

Calculates the SINR values (in linear scale) of all streams of all users with the current IA solution.

The noise variance used will be the value of the noise_var property.

Parameters

- **F** (*list[np.ndarray] | np.ndarray*) – The precoders of all users. This should be either a list of numpy 2D arrays or a 1D numpy array of 2D numpy arrays.
- **U** (*list[np.ndarray] | np.ndarray*) – The receive filters of all users. This should be either a list of numpy 2D arrays or a 1D numpy array of 2D numpy arrays.
- **pe** (*float*) – Power of the external interference source.

Returns **SINRs** – The SINR (in linear scale) of all streams of all users. This is a 1D numpy array of 1D numpy arrays (of floats)

Return type np.ndarray

calc_cov_matrix_extint_plus_noise(*pe: float = 1.0*) → numpy.ndarray

Calculates the covariance matrix of the external interference plus noise.

Parameters **pe** (*float, optional [default=1]*) – External interference power (in linear scale)

Returns **R_all_k** – Return a numpy array, where each element is the covariance matrix of the external interference plus noise at one receiver.

Return type np.ndarray

calc_cov_matrix_extint_without_noise(*pe: float = 1.0*) → numpy.ndarray
Calculates the covariance matrix of the external interference without include the noise.

Parameters **pe** (*float, optional*) – External interference power (in linear scale)

Returns **R_all_k** – Return a numpy array, where each element is the covariance matrix of the external interference at one receiver.

Return type np.ndarray

corrupt_concatenated_data(*data: numpy.ndarray*) → numpy.ndarray
Corrupt data passed through the channel.

If the noise_var member variable is not None then an white noise will also be added.

Parameters **data** (*np.ndarray*) – A bi-dimensional numpy array with the concatenated data of all transmitters as well as the data from all external interference sources. The dimension of data is (sum(self._Nt) + sum(self.extIntNt)) x NSymb. That is, the number of rows corresponds to the sum of the number of transmit antennas of all users and external interference sources and the number of columns correspond to the number of transmitted symbols.

Returns **output** – A bi-dimension numpy array where the number of rows corresponds to the sum of the number of receive antennas of all users and the number of columns correspond to the number of transmitted symbols.

Return type np.ndarray

corrupt_data(*data: numpy.ndarray*, *ext_int_data: numpy.ndarray*) → numpy.ndarray
Corrupt data passed through the channel.

If the noise_var member variable is not None then an white noise will also be added.

Parameters

- **data** (*np.ndarray*) – An array of numpy matrices with the data of the multiple users. The k-th element in *data* is a numpy array with dimension Nt_k x NSymbs, where Nt_k is the number of transmit antennas of the k-th user and NSymbs is the number of transmitted symbols.
- **ext_int_data** (*list[np.ndarray] | np.ndarray*) – An array of numpy matrices with the data of the external interference sources. The l-th element is the data transmitted by the l-th external interference source, which must have a dimension of NtEl x NSymbs, where NtEl is the number of transmit antennas of the l-th external interference source.

Returns **output** – A numpy array where each element contains the received data (a 2D numpy array) of a user.

Return type np.ndarray

property extIntK
Get method for the extIntK property.

property extIntNt
Get method for the extIntNt property.

get_Hk_with_ext_int(*k: int*) → numpy.ndarray
Get the channel from all transmitters (including the external interference sources) to receiver *k*.

This method is essentially the same as the get_Hk method.

Parameters **k** (*int*) – Receiving user.

Returns **channel_k** – Channel from all transmitters to receiver *k*.

Return type np.ndarray

See also:

get_Hkl(), *get_Hk()*, *get_Hk_without_ext_int()*

Examples

```
>>> multiH = MultiUserChannelMatrixExtInt()
>>> H = np.reshape(np.r_[0:20], [4,5])
>>> Nt = np.array([2, 2])
>>> Nr = np.array([2, 2])
>>> NextInt = np.array([1])
>>> multiH.init_from_channel_matrix(H, Nr, Nt, 2, 1)
>>> # Note that the last column of multiH.big_H corresponds to the
>>> # external interference source
>>> print(multiH.big_H)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
>>> print(multiH.get_Hk_with_ext_int(0))
[[0 1 2 3 4]
 [5 6 7 8 9]]
>>> print(multiH.get_Hk_with_ext_int(1))
[[10 11 12 13 14]
 [15 16 17 18 19]]
```

get_Hk_without_ext_int(*k: int*) → numpy.ndarray

Get the channel from all transmitters (without including the external interference sources) to receiver *k*.

Parameters **k** (*int*) – Receiving user.

Returns **channel_k** – Channel from all transmitters to receiver *k* (2D numpy array).

Return type np.ndarray

See also:

get_Hkl(), *get_Hk()*, *get_Hk_with_ext_int()*

Examples

```
>>> multiH = MultiUserChannelMatrixExtInt()
>>> H = np.reshape(np.r_[0:20], [4,5])
>>> Nt = np.array([2, 2])
>>> Nr = np.array([2, 2])
>>> NextInt = np.array([1])
>>> multiH.init_from_channel_matrix(H, Nr, Nt, 2, 1)
>>> # Note that the last column of multiH.big_H corresponds to the
>>> # external interference source
>>> print(multiH.big_H)
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]]
>>> print(multiH.get_Hk_without_ext_int(0))
```

(continues on next page)

(continued from previous page)

```
[[0 1 2 3]
 [5 6 7 8]]
>>> print(multiH.get_Hk_without_ext_int(1))
[[10 11 12 13]
 [15 16 17 18]]
```

init_from_channel_matrix(*channel_matrix: numpy.ndarray*, *Nr: numpy.ndarray*, *Nt: numpy.ndarray*, *K: int*, *NtE: Iterable[int]*) → None

Initializes the multiuser channel matrix from the given *channel_matrix*.

Note that *channel_matrix* must also include the channel terms for the external interference, which must be the last *NtE* columns of *channel_matrix*. The number of rows in *channel_matrix* must be equal to np.sum(Nr), while the number of columns must be np.sum(Nt) + NtE.

Parameters

- **channel_matrix** (*np.ndarray*) – A matrix concatenating the channel of all transmitters (including the external interference sources) to all receivers.
- **Nr** (*np.ndarray*) – Number of antennas at each receiver.
- **Nt** (*np.ndarray*) – Number of antennas at each transmitter.
- **K** (*int*) – Number of transmit/receive pairs.
- **NtE** (*int | list[int] | np.ndarray*) – Number of transmit antennas of the external interference source(s). If NtE is an iterable, the number of external interference sources will be the len(NtE).

Raises **ValueError** – If the arguments are invalid.

randomize(*Nr: Union[numpy.ndarray, int]*, *Nt: Union[numpy.ndarray, int]*, *K: int*, *NtE: Iterable[int]*) → None

Generates a random channel matrix for all users as well as for the external interference source(s).

Parameters

- **Nr** (*int | np.ndarray*) – Number of receive antennas of each user. If an integer is specified, all users will have that number of receive antennas.
- **Nt** (*int | np.ndarray*) – Number of transmit antennas of each user. If an integer is specified, all users will have that number of receive antennas.
- **K** (*int*) – Number of users.
- **NtE** (*int | list[int] | np.ndarray*) – Number of transmit antennas of the external interference source(s). If NtE is an iterable, the number of external interference sources will be the len(NtE).

set_pathloss(*pathloss_matrix: Optional[numpy.ndarray] = None*, *ext_int_pathloss: Optional[numpy.ndarray] = None*) → None

Set the path loss (IN LINEAR SCALE) from each transmitter to each receiver, as well as the path loss from the external interference source(s) to each receiver.

The path loss will be accounted when calling the get_Hkl, get_Hk, the corrupt_concatenated_data and the corrupt_data methods.

Note that path loss is a power relation, which means that the channel coefficients will be multiplied by the square root of elements in *pathloss_matrix*.

If you want to disable the path loss, set pathloss_matrix to None.

Parameters

- **pathloss_matrix** (*np.ndarray*) – A matrix with dimension "K x K", where K is the number of users, with the path loss (IN LINEAR SCALE) from each transmitter (columns) to each receiver (rows). If you want to disable the path loss then set it to None. ext_int_pathloss : 2D numpy array The path loss from each interference source to each receiver. The number of rows of ext_int_pathloss must be equal to the number of receives, while the number of columns must be equal to the number of external interference sources.
- **ext_int_pathloss** (*np.ndarray*) – The external interference path loss.

pyphysim.channels.noise module

pyphysim.channels.noise.**calc_thermal_noise_power_dBm**(*T: float*, *delta_f: float*) → float

Calculate the thermal noise power (in dBm) for the given room temperature *T* (in Cº) and bandwidth *delta_f* (in Hz).

Parameters

- **T** (*float*) – Room temperature in Cesium degrees.
- **delta_f** (*float*) – Bandwidth in Hz.

Returns **noise_var** – The noise power.

Return type float

pyphysim.channels.pathloss module

Implement classes for several Path loss models.

The *PathLossBase* class implements the common code to every path loss model and only two methods need to be implemented in subclasses: the *PathLossBase.which_distance_dB()* and the *PathLossBase._calc_deterministic_path_loss_dB()* methods. However, instead of inheriting directly from *PathLossBase*, inherit from either *PathLossIndoorBase* or *PathLossOutdoorBase*.

The most common usage of a path loss class is to instantiate an object of the desired path loss model and then call the **calc_path_loss_dB** or the **calc_path_loss** methods to actually calculate the path loss.

class pyphysim.channels.pathloss.**PathLoss3GPP1**

Bases: *pyphysim.channels.pathloss.PathLossGeneral*

Class to calculate the Path Loss according to the model from 3GPP (scenario 1). That is, the Path Loss (in dB) is equal to

$$PL = 128.1 + 37.6 * \log 10(d)$$

This model is valid for LTE assumptions and at 2GHz frequency, where the distance is in Km.

Examples

Determining the path loss in the free space for a distance of 1Km (without considering shadowing).

```
>>> pl = PathLoss3GPP1()
>>> pl.calc_path_loss(1)        # linear scale
1.5488166189124858e-13
>>> pl.calc_path_loss_dB(1)     # log scale
128.1
```

Determining the distance (in Km) that yields a path loss of 130dB.

```
>>> pl.which_distance_dB(130)
1.1233935211892188
```

_repr_latex_() → str
Get a Latex representation of the PathLossFreeSpace class.

This is useful for representing the path loss object in an IPython notebook.

Returns The Latex representation of the path loss object.

Return type str

class pyphysim.channels.pathloss.**PathLossBase**
Bases: object

Base class for the different Path Loss models.

The common interface for the path loss classes is provided by the *calc_path_loss_dB()* or the *calc_path_loss()* methods to actually calculate the path loss for a given distance, as well as the *which_distance_dB()* or *which_distance()* methods to determine the distance that yields the given path loss.

Each subclass of PathLossBase NEED TO IMPLEMENT only the *which_distance_dB()* and the *_calc_deterministic_path_loss_dB()* functions.

If the *use_shadow_bool* attribute is set to True then calling *calc_path_loss_dB()* or *calc_path_loss()* will take the shadowing specified in the *sigma_shadow* attribute into account. However, shadowing is not taken into account in the *which_distance_dB()* and *which_distance()* functions, regardless of the value of the *use_shadow_bool* variable.

sigma_shadow
The shadowing (in dB)

Type int

use_shadow_bool
If True then shadowing will be used.

Type bool

handle_small_distances_bool
If this is True then any negative path loss (in dB) that appears because a distance is too small will considered as 0dB. If this s False then an exception will be raised instead.

Type bool

_TYPE = 'base'

abstract _calc_deterministic_path_loss_dB(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray
Calculates the Path Loss (in dB) for a given distance (in Km) without including the shadowing.

Parameters

- **d** (*float | np.ndarray*) – Distance (in Km)
- **kargs** (*dict, optional*) – Optional parameters for use in subclasses if required.

Other Parameters **kwargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **PL** – Path loss (in dB).

Return type float | np.ndarray

Raises **NotImplementedError** – If the _calc_deterministic_path_loss_dB method of the PathLossBase class is called.

_plot_deterministic_path_loss_in_dB_impl(*d: numpy.ndarray*, *ax: Optional[Any] = None*, *extra_args: Optional[Any] = None*, *distance_unit: str = 'Km'*) → None

Plot the path loss (in dB) for the distance values in *d* (in Km).

Parameters

- **d** (*np.ndarray*) – Distance (in correct unit)
- **ax** (*A matplotlib ax, optional*) – The ax where the path loss will be plotted. If not provided, a new figure (and ax) will be created.
- **extra_args** (*dict*) – Extra arguments that will be passed to the ax.plot command as `**extra_args` (see Matplotlib documentation). Ex: {'label': 'curve name', 'linewidth': 2}

calc_path_loss(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray

Calculates the path loss (linear scale) for a given distance (in Km).

Parameters

- **d** (*float | np.ndarray*) – Distance (in Km)
- **kargs** (*dict*) – Extra named parameters. This is used in subclasses for extra parameters for the path loss calculation.

Other Parameters **kwargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **pl** – Path loss (in linear scale) for the given distance(s).

Return type float | np.ndarray

calc_path_loss_dB(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray

Calculates the Path Loss (in dB) for a given distance (in Km).

Note that the returned value is positive, but should be understood as "a loss".

Parameters

- **d** (*float | np.ndarray*) – Distance (in Km)
- **kargs** (*dict*) – Optional parameters for use in subclasses if required.

Other Parameters **kwargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **PL** – Path loss (in dB) for the given distance(s).

Return type float | np.ndarray

plot_deterministic_path_loss_in_dB(*d: numpy.ndarray*, *ax: Optional[Any] = None*, *extra_args: Optional[Any] = None*) → None

Plot the path loss (in dB) for the distance values in *d* (in Km).

Parameters

- **d** (*np.ndarray*) – Distance (in Km)
- **ax** (*A matplotlib ax, optional*) – The ax where the path loss will be plotted. If not provided, a new figure (and ax) will be created.
- **extra_args** (*dict*) – Extra arguments that will be passed to the ax.plot command as `**extra_args` (see Matplotlib documentation). Ex: {'label': 'curve name', 'linewidth': 2}

property type
Get method for the type property.

which_distance (*pl: NumberOrArray*) → NumberOrArray
Calculates the required distance (in Km) to achieve the given path loss. It is the inverse of the calc_path_loss function.

Parameters **pl** (*float | np.ndarray*) – Path loss (in linear scale).

Returns **d** – Distance(s) that will yield the path loss *pl*.

Return type float | np.ndarray

abstract which_distance_dB (*PL: NumberOrArray*) → NumberOrArray
Calculates the distance that yields the given path loss (in dB).

Parameters **PL** (*float | np.ndarray*) – Path Loss (in dB)

Returns **d** – Distance to get the desired path loss *PL*.

Return type float | np.ndarray

Raises **NotImplementedError** – If the which_distance_dB method of the PathLossBase class is called.

class pyphysim.channels.pathloss.**PathLossFreeSpace** (*n: float = 2.0, fc: float = 900.0*)
Bases: *pyphysim.channels.pathloss.PathLossGeneral*

Class to calculate the Path Loss in the free space.

The common interface for the path loss classes is provided by the *PathLossOutdoorBase.calc_path_loss_dB()* or the *PathLossOutdoorBase.calc_path_loss()* methods to actually calculate the path loss for a given distance, as well as the *PathLossGeneral.which_distance_dB()* or *PathLossBase.which_distance()* methods to determine the distance that yields the given path loss.

For the path loss in free space you also need to set the *n* variable, corresponding to the path loss coefficient, and the *fc* variable, corresponding to the frequency. The *n* variable defaults to 2 and *fc* defaults to 900 (that is, 900MHz).

The path loss (in dB) in free space is calculated as:

$$PL = 10n(\log_{10}(d) + \log_{10}(fc * 1e6) - 4.3779113907)$$

Likewise, the *PathLossGeneral.which_distance_dB()* function calculates the value of

$$10^{(PL/(10n) - \log_{10}(fc) + 4.377911390697565)}$$

Parameters

- **n** (*float*) – Path loss exponent.
- **fc** (*float*) – Central carrier frequency (in MHz).

Examples

Determining the path loss in the free space for a distance of 1Km (without considering shadowing).

```
>>> pl = PathLossFreeSpace()
>>> pl.calc_path_loss(1)        # linear scale
7.036193308495632e-10
>>> pl.calc_path_loss_dB(1)     # log scale
91.5266223748352
```

Determining the distance (in Km) that yields a path loss of 90dB.

```
>>> pl.which_distance_dB(90)
0.8388202017414481
```

static _calculate_C_from_fc_and_n (*fc: float*, *n: float*) → float
Calculate the value of the constant *C* for the frequency value *fc* and path loss exponent *n*.

Parameters

- **fc** (*float*) – Central carrier frequency (in MHz)
- **n** (*float*) – Path loss exponent

Returns **C** – Constant *C* for the Free Space path loss model for the given frequency and path loss exponent.

Return type float

_repr_latex_ () → str
Get a Latex representation of the PathLossFreeSpace class.

This is useful for representing the path loss object in an IPython notebook.

Returns The Latex representation of the path loss object.

Return type str

property fc
Get the central carrier frequency.

Returns The central carrier frequency.

Return type float

property n
Get method for the n property.

Returns The path loss exponent.

Return type float

class pyphysim.channels.pathloss.**PathLossGeneral** (*n: float*, *C: float*)
Bases: *pyphysim.channels.pathloss.PathLossOutdoorBase*

Class to calculate the path loss given the path loss for a reference distance.

In its simplest form, the path loss can be calculated using the formula

$PL = 10n\log_{10}(d) + C$

where *PL* is in dB, *n* is the path loss exponent (usually in the range of 2 to 4) and *d* is the distance between the transmitter and the receiver.

Parameters

- **n** (*float*) – The path loss exponent.
- **C** (*float*) – The constant *C* in the path loss formula.

_calc_deterministic_path_loss_dB (*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray
Calculates the Path Loss (in dB) for a given distance (in Km).

Note that the returned value is positive, but should be understood as "a loss".

For d in Km and self.fc in MHz, the free space Path Loss is given by

$$PL = 10n\log_{10}(d) + C$$

Parameters **d** (*float | np.ndarray*) – Distance (in Km).

Returns **pl_dB** – Path loss in dB.

Return type float | np.ndarray

_get_latex_repr() → str
Get the Latex representation (equation) for the PathLossGeneral class.

The general equation is given by

$$PL = 10n \log_{10}(d) + C$$

Returns The Latex representation of the path loss object.

Return type str

_repr_latex_() → str
Get a Latex representation of the PathLossGeneral class.

This is useful for representing the path loss object in an IPython notebook.

Returns The Latex representation of the path loss object.

Return type str

which_distance_dB(*PL: NumberOrArray*) → NumberOrArray
Calculates the required distance (in Km) to achieve the given path loss (in dB).

It is the inverse of the calc_path_loss function.

$$10^{(PL/(10n)-C)}$$

d = obj.whichDistance(dB2Linear(-PL));

Parameters **PL** (*float | np.ndarray*) – Path Loss (in dB).

Returns **d** – Distance (in Km).

Return type float | np.ndarray

class pyphysim.channels.pathloss.**PathLossIndoorBase**
Bases: *pyphysim.channels.pathloss.PathLossBase*

Base class for the different Indoor Path Loss models.

The common interface for the path loss classes is provided by the *calc_path_loss_dB()* or the *calc_path_loss()* methods to actually calculate the path loss for a given distance, as well as the *which_distance_dB()* or which_distance methods to determine the distance that yields the given path loss.

Each subclass of PathLossBase NEED TO IMPLEMENT only the *which_distance_dB()* and the *_calc_deterministic_path_loss_dB()* functions.

If the *use_shadow_bool* is set to True then calling *calc_path_loss_dB()* or *calc_path_loss()* will take the shadowing specified in the *sigma_shadow* variable into account. However, shadowing is not taken into account in the *which_distance_dB()* and *which_distance()* functions, regardless of the value of the *use_shadow_bool* variable.

_TYPE = 'indoor'

abstract _calc_deterministic_path_loss_dB(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray
Calculates the Path Loss (in dB) for a given distance (in meters) without including the shadowing.

Parameters **d** (*float | np.ndarray*) – Distance (in meters)

Other Parameters **kwargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **PL** – The calculated path loss.

Return type float | np.ndarray

Raises **NotImplementedError** – If the _calc_deterministic_path_loss_dB method of the PathLossBase class is called.

calc_path_loss(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray
Calculates the path loss (linear scale) for a given distance (in meters).

Parameters

- **d** (*float | np.ndarray*) – Distance (in meters)
- **kargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **pl** – Path loss (in linear scale) for the given distance(s).

Return type float | np.ndarray

calc_path_loss_dB(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray
Calculates the Path Loss (in dB) for a given distance (in meters).

Note that the returned value is positive, but should be understood as "a loss".

Parameters

- **d** (*float | np.ndarray*) – Distance (in meters)
- **kargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **PL** – Path loss (in dB) for the given distance(s).

Return type float | np.ndarray

plot_deterministic_path_loss_in_dB(*d: NumberOrArray*, *ax: Optional[Any] = None*, *extra_args: Optional[Any] = None*) → None
Plot the path loss (in dB) for the distance values in *d* (in meters).

Parameters

- **d** (*np.ndarray*) – Distance (in meters)
- **ax** (*A matplotlib ax, optional*) – The ax where the path loss will be plotted. If not provided, a new figure (and ax) will be created.
- **extra_args** (*dict*) – Extra arguments that will be passed to the ax.plot command as `**extra_args` (see Matplotlib documentation). Ex: {'label': 'curve name', 'linewidth': 2}

which_distance(*pl: NumberOrArray*) → NumberOrArray
Calculates the required distance (in meters) to achieve the given path loss. It is the inverse of the calc_path_loss function.

Parameters **pl** (*float | np.ndarray*) – Path loss (in linear scale).

Returns **d** – Distance(s) that will yield the path loss *pl*.

Return type float | np.ndarray

abstract which_distance_dB(*PL: NumberOrArray*) → NumberOrArray
Calculates the distance that yields the given path loss (in dB).

Parameters **PL** (*float | np.ndarray*) – Path Loss (in dB)

Returns **d** – The distance to yield the given path loss.

Return type float | np.ndarray

Raises **NotImplementedError** – If the which_distance_dB method of the PathLossBase class is called.

class pyphysim.channels.pathloss.**PathLossMetisPS7**(*fc: float = 900.0*)

Bases: *pyphysim.channels.pathloss.PathLossIndoorBase*

Class to calculate the Path Loss (indoor) according to the model described for the Propagation Scenario (PS) 7 of the METIS project.

This model is an indoor-2-indoor model.

_calc_PS7_path_loss_dB_LOS_same_floor(*d: NumberOrArray*) → NumberOrArray

Calculate the deterministic path loss according to the Propagation Scenario (PS) 7 of the METIS project for the LOS case.

The path loss (in dB) is calculated as

$$PL = A\log_{10}(d) + B + C\log_{10}(f_c/5)$$

The distance d is in meters, while the frequency f_c is in GHz.

The other variables (NLOS case) are:

$$A = 18.7 B = 46.8, C = 20$$

where n_w is the number of walls between the transmitter and receiver.

Parameters **d** (*float | np.ndarray*) – Distance (in meters).

Returns **pl_dB** – Path loss in dB.

Return type float | np.ndarray

_calc_PS7_path_loss_dB_NLOS_same_floor(*d: NumberOrArray*, *num_walls: int = 1*) → NumberOrArray

Calculate the deterministic path loss according to the Propagation Scenario (PS) 7 of the METIS project for the NLOS case.

The path loss (in dB) is calculated as

$$PL = A\log_{10}(d) + B + C\log_{10}(f_c/5) + X$$

The distance d is in meters, while the frequency f_c is in Hz.

The other variables (NLOS case) are:

$$A = 36.8 B = 43.8, C = 20, X = 5(n_w - 1)$$

where n_w is the number of walls between the transmitter and receiver.

Parameters

- **d** (*float | np.ndarray*) – Distance (in meters).
- **num_walls** (*int | np.ndarray*) – Number of walls between the transmitter and the receiver. If it is an int array it must have the same dimension as *d*.

Returns **pl_dB** – Path loss in dB.

Return type float | np.ndarray

_calc_PS7_path_loss_dB_same_floor(*d: NumberOrArray*, *num_walls: int = 0*) → NumberOrArray

Calculate the deterministic path loss according to the Propagation Scenario (PS) 7 of the METIS project.

The path loss (in dB) is calculated as

$$PL = A\log_{10}(d) + B + C\log_{10}(f_c/5) + X$$

The distance d is in meters, while the frequency f_c is in GHz. Na others variables a different for the LOS and NLOS cases.

For the LOS case we have:

$$A = 18.7 B = 46.8, C = 20, X = 0$$

For the NLOS case we have:

$$A = 36.8 B = 43.8, C = 20, X = 5(n_w - 1)$$

where n_w is the number of walls between the transmitter and receiver.

Parameters

- **d** (*float | np.ndarray*) – Distance (in meters).
- **num_walls** (*int | np.ndarray*) – Indicates how many walls the signal has to pass. If num_walls is zero, then Line-of-Sight parameters are used. If it is greater than zero then Non-Sign-of-Sight parameters are used. If num_walls is a numpy array then it must have the same dimension as *d* and it then specifies the number of walls for each individual value of *d*

Returns **pl_dB** – Path loss in dB.

Return type float | np.ndarray

_calc_deterministic_path_loss_dB(*d: NumberOrArray*, *num_walls: int = 0*) → NumberOrArray

Calculates the Path Loss (in dB) for a given distance (in meters) without including the shadowing.

Parameters

- **d** (*float | np.ndarray*) – Distance (in meters)
- **num_walls** (*int | np.ndarray*) – Number of walls between the transmitter and the receiver. If it is an int array it must have the same dimension as *d*.

Returns **PL** – The calculated path loss.

Return type float | np.ndarray

_repr_latex_() → str

Get a Latex representation of the PathLossMetisPS7 class.

This is useful for representing the path loss object in an IPython notebook.

Returns The Latex representation of the path loss object.

Return type str

property fc

Get the central carrier frequency.

Returns The central carrier frequency.

Return type float

static get_latex_repr(*num_walls: Optional[int] = None*) → str
Get the Latex representation (equation) for the PathLossGeneral class.

The general equation is given by

$$PL = A\log_{10}(d) + B + C\log_{10}(f_c/5) + X$$

where the parameters A, B, C and X depend on the number of walls.

Parameters **num_walls** (*int, None*) – Number of walls. LOS is used if it is 0 and NLOS is used if it is greater than zero. If it is None, then letters are used instead of numeric values.

Returns The Latex representation (equation) for the PathLossGeneral class.

Return type str

which_distance_dB(*PL: NumberOrArray*) → NumberOrArray
Calculates the distance that yields the given path loss (in dB).

Parameters **PL** (*float | np.ndarray*) – Path Loss (in dB)

Returns **d** – The distance to yield the given path loss.

Return type float | np.ndarray

Raises **NotImplementedError** – If the which_distance_dB method of the PathLossBase class is called.

class pyphysim.channels.pathloss.**PathLossOkomuraHata**
Bases: *pyphysim.channels.pathloss.PathLossOutdoorBase*

Class to calculate the Path Loss according to the Okomura Hata model.

The exact formula depend on the area type, but in general the path loss is given by (in dB):

$$PL = 69.55 + 26.16 * \log(fc) - 13.82 * \log(h_{bs}) - a(h_{ms}) + (44.9 - 6.55\log(h_{bs}))\log(d) - K$$

The term $a(h_{ms})$ is the mobile station correction factor (see *_calc_mobile_antenna_height_correction_factor()*

The term K is a correction factor that depends on the area type (see *_calc_K()*).

The possible area types are (in ascending order): 'open', 'suburban', 'medium city' and 'large city'.

_calc_K() → float
Calculates the "'medium city'/'suburban'/'open area'" correction factor.

Returns **K** – The correction factor "K".

Return type float

_calc_deterministic_path_loss_dB(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray
Calculates the Path Loss (in dB) for a given distance (in Km).

Note that the returned value is positive, but should be understood as "a loss".

For d in Km and self.fc in Hz, the free space Path Loss is given by

$$PL = 10n(\log_{10}(d) + \log_{10}(f) - 4.3779113907)$$

Parameters **d** (*float | np.ndarray*) – Distance (in Km). Can be between 1km and 20Km.

Returns **PL** – Path loss in dB.

Return type float | np.ndarray

_calc_mobile_antenna_height_correction_factor() → float
Calculates the mobile antenna height correction factor.

This factor is strongly impacted by surrounding buildings and is refined according to city sizes. For all area types, except 'large city', the mobile antenna correction factor is given by

$$a(h_{ms}) = (1.1\log(f) - 0.7)h_{ms} - 1.56\log(f) + 0.8$$

For the 'large city' area type, the mobile antenna height correction is given by

$$a(h_{ms}) = 3.2(\log(11.75 * h_{ms})^2) - 4.97$$

if the frequency is greater then 300MHz, or

$$a(h_{ms}) = 8.29(\log(1.54h_{ms}))^2 - 1.10$$

if the frequency is lower than 300MHz (and greater than 150MHz where the Okomura Hata model is valid).

Returns The mobile antenna height correction.

Return type float

property area_type
Get method for the area_type property.

property fc
Get the central carrier frequency.

Returns The central carrier frequency.

Return type float

property hbs
Get the height of the Base Station property.

Returns Height of the Base Station (in meters).

Return type float

property hms
Get the height of the Mobile Station property.

Returns Height of the Mobile Station (in meters).

Return type float

which_distance_dB(*PL: NumberOrArray*) → NumberOrArray
Calculates the required distance (in Km) to achieve the given path loss (in dB).

It is the inverse of the calc_path_loss function.

Parameters **PL** (*float | np.ndarray*) – Path loss (in dB).

Returns **d** – Distance (in Km).

Return type float | np.ndarray

class pyphysim.channels.pathloss.**PathLossOutdoorBase**
Bases: *pyphysim.channels.pathloss.PathLossBase*

Base class for the different Outdoor Path Loss models.

The common interface for the path loss classes is provided by the *calc_path_loss_dB()* or the *calc_path_loss()* methods to actually calculate the path loss for a given distance, as well as the

which_distance_dB() or *PathLossBase.which_distance()* methods to determine the distance that yields the given path loss.

Each subclass of PathLossBase NEED TO IMPLEMENT only the *which_distance_dB()* and the *_calc_deterministic_path_loss_dB()* functions.

If the *use_shadow_bool* is set to True then calling *calc_path_loss_dB()* or *calc_path_loss()* will take the shadowing specified in the *sigma_shadow* variable into account. However, shadowing is not taken into account in the *which_distance_dB()* and *PathLossBase.which_distance()* functions, regardless of the value of the *use_shadow_bool* variable.

_TYPE = 'outdoor'

abstract _calc_deterministic_path_loss_dB(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray

Calculates the Path Loss (in dB) for a given distance (in Km) without including the shadowing.

Parameters **d** (*float | np.ndarray*) – Distance (in Km)

Other Parameters **kwargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **PL** – The calculated path loss (in dB).

Return type float | np.ndarray

Raises **NotImplementedError** – If the _calc_deterministic_path_loss_dB method of the PathLossBase class is called.

calc_path_loss(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray

Calculates the path loss (linear scale) for a given distance (in Km).

Parameters

- **d** (*float | np.ndarray | list[float]*) – Distance (in Km)
- **kargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **pl** – Path loss (in linear scale) for the given distance(s).

Return type float | np.ndarray

calc_path_loss_dB(*d: NumberOrArray*, ***kargs: Any*) → NumberOrArray

Calculates the Path Loss (in dB) for a given distance (in Km).

Note that the returned value is positive, but should be understood as "a loss".

Parameters

- **d** (*float | np.ndarray | list[float]*) – Distance (in Km)
- **kargs** (*dict*) – Additional keywords that might be necessary in a subclass.

Returns **PL** – Path loss (in dB) for the given distance(s).

Return type float | np.ndarray

plot_deterministic_path_loss_in_dB(*d: NumberOrArray*, *ax: Optional[Any] = None*, *extra_args: Optional[Any] = None*) → None

Plot the path loss (in dB) for the distance values in `d` (in Km).

Parameters

- **d** (*np.ndarray*) – Distance (in Km)
- **ax** (*A matplotlib ax, optional*) – The ax where the path loss will be plotted. If not provided, a new figure (and ax) will be created.

- **extra_args** (*dict*) – Extra arguments that will be passed to the ax.plot command as `**extra_args` (see Matplotlib documentation). Ex: {'label': 'curve name', 'linewidth': 2}

abstract which_distance_dB(*PL: NumberOrArray*) → NumberOrArray
Calculates the distance that yields the given path loss (in dB).

Parameters **PL** (*float | np.ndarray*) – Path Loss (in dB)

Returns **d** – The distance that yields the given path loss.

Return type float | np.ndarray

Raises **NotImplementedError** – If the which_distance_dB method of the PathLossBase class is called.

pyphysim.channels.singleuser module

Module containing single user channels.

class pyphysim.channels.singleuser.**SuChannel**(*fading_generator: Optional[Union[*pyphysim.channels.fading_generators.JakesSampleGe pyphysim.channels.fading_generators.RayleighSampleGenerator*]] = None, channel_profile: Optional[*pyphysim.channels.fading.TdlChannelProfile*] = None, tap_powers_dB: Optional[*numpy.ndarray*] = None, tap_delays: Optional[*numpy.ndarray*] = None, Ts: Optional[*float*] = None*)

Bases: object

Single User channel corresponding to a Tapped Delay Line channel model, which corresponds to a multipath channel. You can use a single tap in order to get a flat fading channel.

You can create a new SuChannel object either specifying the channel profile or specifying both the channel tap powers and delays. If only the fading_generator is specified then a single tap with unitary power and delay zero will be assumed, which corresponds to a flat fading channel model.

Parameters

- **fading_generator** (*FadingGenerator*) – The instance of a fading generator in the *fading_generators* module. It should be a subclass of FadingSampleGenerator. The fading generator will be used to generate the channel samples. If not provided then RayleighSampleGenerator will be used
- **channel_profile** (fading.TdlChannelProfile) – The channel profile, which specifies the tap powers and delays.
- **tap_powers_dB** (*np.ndarray*) – The powers of each tap (in dB). Dimension: *L x 1* Note: The power of each tap will be a negative number (in dB).
- **tap_delays** (*np.ndarray*) – The delay of each tap (in seconds). Dimension: *L x 1*
- **Ts** (*float, optional*) – The sampling interval.

property channel_profile
Return the channel profile.

Returns The channel profile.

Return type *fading.TdlChannelProfile*

corrupt_data(*signal: numpy.ndarray*) → numpy.ndarray
Transmit the signal through the TDL channel.

Parameters **signal** (*np.ndarray*) – The signal to be transmitted.

Returns The received signal after transmission through the TDL channel.

Return type np.ndarray

corrupt_data_in_freq_domain(*signal: numpy.ndarray*, *fft_size: int*, *carrier_indexes: Optional[Union[numpy.ndarray, List[int], slice]] = None*) → numpy.ndarray
Transmit the signal through the TDL channel, but in the frequency domain.

This is ROUGHLY equivalent to modulating *signal* with OFDM using *fft_size* subcarriers, transmitting through a regular TdlChannel, and then demodulating with OFDM to recover the received signal.

One important difference is that here the channel is considered constant during the transmission of *fft_size* elements in *signal*, and then it is varied by the equivalent of the variation for that number of elements. That is, the channel is block static.

Parameters

- **signal** (*np.ndarray*) – The signal to be transmitted.
- **fft_size** (*int*) – The size of the Fourier transform to get the frequency response.
- **carrier_indexes** (*slice | np.ndarray*) – The indexes of the subcarriers where signal is to be transmitted. If it is None assume all subcarriers will be used. This can be a slice object or a numpy array of integers.

Returns The received signal after transmission through the TDL channel

Return type np.ndarray

get_last_impulse_response() → *pyphysim.channels.fading.TdlImpulseResponse*
Get the last generated impulse response.

A new impulse response is generated when the method *corrupt_data* is called. You can use the *get_last_impulse_response* method to get the impulse response used to corrupt the last data.

Returns The impulse response of the channel that was used to corrupt the last data.

Return type *fading.TdlImpulseResponse*

property num_rx_antennas
Get the number of receive antennas.

Returns The number of receive antennas.

Return type int

property num_taps
Get the number of taps in the profile.

Returns The number of taps in the channel (not including any zero padding).

Return type int

property num_taps_with_padding
Get the number of taps in the profile including zero-padding when the profile is discretized.

If the profile is not discretized an exception is raised.

Returns The number of taps in the channel (including any zero padding).

Return type int

property num_tx_antennas
Get the number of transmit antennas.

Returns The number of transmit antennas.

Return type int

set_num_antennas(*num_rx_antennas: int*, *num_tx_antennas: int*) → None
Set the number of transmit and receive antennas for MIMO transmission.

Set both *num_rx_antennas* and *num_tx_antennas* to None for SISO transmission

Parameters

- **num_rx_antennas** (*int*) – The number of receive antennas.
- **num_tx_antennas** (*int*) – The number of transmit antennas.

set_pathloss(*pathloss_value: Optional[float] = None*) → None
Set the path loss (IN LINEAR SCALE).

The path loss will be accounted when calling the corrupt_data method.

If you want to disable the path loss, set *pathloss_value* to None.

Parameters **pathloss_value** (*float | None*) – The path loss (IN LINEAR SCALE) from the transmitter to the receiver. If you want to disable the path loss then set it to None.

Notes

Note that path loss is a power relation, which means that the channel coefficients will be multiplied by the square root of elements in *pathloss_value*.

property switched_direction
Get the value of *switched_direction*.

Returns True if direction is switched and False otherwise.

Return type bool

class pyphysim.channels.singleuser.**SuMimoChannel**(*num_antennas: int*, *fading_generator: Optional[Union[*pyphysim.channels.fading_generators.JakesSam
pyphysim.channels.fading_generators.RayleighSampleGenera
= None, *channel_profile: Optional[*pyphysim.channels.fading.TdlChannelProfile*] = None*, *tap_powers_dB: Optional[numpy.ndarray] = None*, *tap_delays: Optional[numpy.ndarray] = None*, *Ts: Optional[float] = None*)
Bases: *pyphysim.channels.singleuser.SuChannel*

Single User channel corresponding to a Tapped Delay Line channel model, which corresponds to a multipath channel. You can use a single tap in order to get a flat fading channel.

You can create a new SuMimoChannel object either specifying the channel profile or specifying both the channel tap powers and delays. If only the fading_generator is specified then a single tap with unitary power and delay zero will be assumed, which corresponds to a flat fading channel model.

Parameters

- **num_antennas** (*int*) – Number of transmit and receive antennas.

- **fading_generator** (*FadingGenerator*) – The instance of a fading generator in the *fading_generators* module. It should be a subclass of FadingSampleGenerator. The fading generator will be used to generate the channel samples. If not provided then RayleighSampleGenerator will be used
- **channel_profile** (fading.TdlChannelProfile) – The channel profile, which specifies the tap powers and delays.
- **tap_powers_dB** (*np.ndarray*) – The powers of each tap (in dB). Dimension: *L x 1* Note: The power of each tap will be a negative number (in dB).
- **tap_delays** (*np.ndarray*) – The delay of each tap (in seconds). Dimension: *L x 1*
- **Ts** (*float, optional*) – The sampling interval.

Module contents

Module with implementation of channel related classes.

pyphysim.comm package

Submodules

pyphysim.comm.blockdiagonalization module

Module implementing the block diagonalization algorithm.

There are two ways to use this module. You can either use the *BlockDiagonalizer* class, or you can use the *block_diagonalize()* and the *calc_receive_filter()* functions (which use the BlockDiagonalizer class in their implementation).

class pyphysim.comm.blockdiagonalization.**BDWithExtIntBase**(*num_users: int*, *iPu: float*, *noise_var: float*, *pe: float*)

Bases: *pyphysim.comm.blockdiagonalization.BlockDiagonalizer*

Class to perform the block diagonalization algorithm in a joint transmission scenario taking into account the external interference.

This is the base class for any block diagonalization class that takes into account the external interference.

Parameters

- **num_users** (*int*) – Number of users.
- **iPu** (*float*) – Power available for EACH user (in linear scale).
- **noise_var** (*float*) – Noise variance (power in linear scale).
- **pe** (*float*) – Power of the external interference source (in linear scale)

calc_whitening_matrices(*mu_channel: pyphysim.channels.multiuser.MultiUserChannelMatrixExtInt*) → List[numpy.ndarray]

Calculates the whitening receive filters for each user.

Note that to consider the noise variance it must be set in the provided mu_channel object.

Parameters **mu_channel** (MultiUserChannelMatrixExtInt) – A MultiUserChannelMatrixExtInt object, which has the channel from all the transmitters to all the receivers, as well as th external interference.

Returns **W_all_k** – The whitening matrices that each receiver should use to whiten the external interference. Each element in W_all_k is the whitening filter (with the conjugate transpose already applied) for a receiver.

Return type list[np.ndarray]

class pyphysim.comm.blockdiagonalization.**BlockDiagonalizer**(*num_users: int*, *iPu: float*, *noise_var: float*)

Bases: object

Class to perform the block diagonalization algorithm in a joint transmission scenario.

In the block diagonalization algorithm either a single base station with more antennas transmits for multiple users at the same time or a group of base stations acts as a single transmitter to send data to the multiple users at the same time. In both cases the block diagonalization algorithm assures that each receiver does not see interference from the other receivers.

The water-filling algorithm is also applied to optimally distribute the power. However, in the case with multiple base stations, the power restriction in each base station must be respected. Therefore, after the power is optimally allocated at each base station all powers will be normalized to respect the power restriction of the base transmitting the highest energy. This is what is done in the *block_diagonalize()* method.

If the power should not be optimally allocated with the water-filling algorithm, use the *block_diagonalize_no_waterfilling()* method instead. The power restriction in each base station will still be respected, but the base station will equally divide its power among the available dimensions. Note that the result will be similar to *block_diagonalize()* in the high SNR regime.

Parameters

- **num_users** (*int*) – Number of users.
- **iPu** (*float*) – Power available for EACH user.
- **noise_var** (*float*) – Noise variance (power in linear scale).

Examples

Consider the case where we have 3 base station (BSs) jointly transmitting to 3 users, where each base station has a power of 1.5, the number of antennas (at each BS and at each receiver) is 2, and the noise variance is 1e-4. The channel can be block diagonalized for this scenario with

```
>>> bs_power = 1.5
>>> noise_var = 1e-4
>>> num_users = 2
>>> Ntx = 2  # Number of transmit antennas (per BS)
>>> Nrx = 2  # Number of receive antennas (per user)
>>> # Create the BlockDiagonalizer object
>>> bd = BlockDiagonalizer(num_users, bs_power, noise_var)
>>> channel = np.array([[-0.9834-0.0123j,  0.6503-0.3189j,      0.5484+1.7049j, -
↪1.0891-0.1025j], [-0.5911-0.3055j, -0.6205+0.3375j,     -0.7995+0.3723j,  0.
↪7412-1.2537j], [-0.2732+0.475j , -0.4191+0.4019j,     0.1047-0.5592j,  0.7548-1.
↪0214j], [ 0.5377-0.208j , -0.1480-1.0527j,     -0.6373+0.4081j, -0.5854-0.
↪8135j]])
>>> (newH, Ms) = bd.block_diagonalize(channel)
```

We can see that the equivalent channel (after applying the Ms modulation matrix) is really block diagonalized.

```
>>> print(np.round(newH + 1e-10 + 1e-10j, 4))
[[ 0.0916+0.0135j -1.7449-0.4328j  0.    +0.j      0.    +0.j    ]
 [-0.0114-0.146j   0.0213-1.1366j  0.    +0.j      0.    +0.j    ]
 [ 0.    +0.j      0.    +0.j      0.0868+0.1565j -0.3673+0.2289j]
 [ 0.    +0.j      0.    +0.j     -0.0396+0.0407j  1.024 +0.8997j]]
```

Notice how the power restriction of each BS is respected (although only one BS will transmit with its maximum power).

```
>>> print(np.round(np.linalg.norm(Ms[:,0:Ntx])**2, 4))
1.4997
>>> print(np.round(np.linalg.norm(Ms[:,Ntx:])**2, 4))
1.5
```

Notes

The block diagonalization algorithm is described in [Spencer2004], where different power allocations are illustrated. The *BlockDiagonalizer* class implement two power allocation methods, a global power allocation, and a 'per transmitter' power allocation.

_calc_BD_matrix_no_power_scaling(*mtChannel: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Calculates the modulation matrix "M" that block diagonalizes the channel *mtChannel*, but without any king of power scaling.

The "modulation matrix" is a matrix that changes the channel to a block diagonal structure and it is the first part in the Block Diagonalization algorithm. The returned modulation matrix is equivalent to Equation (12) of [Spencer2004] but without the power scaling matrix Λ. Therefore, for the complete BD algorithm it is still necessary to perform this power scaling in the output of _calc_BD_matrix_no_power_scaling.

Parameters **mtChannel** (*np.ndarray*) – Channel from the transmitter to all users.

Returns **(Ms_bad, Sigma)** – The modulation matrix "Ms_bad" is a precoder that block diagonalizes the channel. The singular values of the equivalent channel when the modulation matrix is applied correspond to Sigma. Therefore, Sigma can be used latter in the power allocation process.

Return type (np.ndarray,np.ndarray)

Notes

The reason why the Block Diagonalization algorithm was broken down into the code here and the power scaling code is because the power scaling may changing depending on the scenario. For instance, if the transmitter corresponds to a single base station the the power may be distributed into all the dimensions of the Modulation matrix. On the other hand, if the transmitter corresponds to multiple base stations jointly transmitting to multiple users then the power of each base station must be distributed only into the dimensions corresponding to that base station.

_get_sub_channel(*mt_channel: numpy.ndarray*, *desired_users: Union[int, Iterable[int]]*) → numpy.ndarray

Get a subchannel according to the desired_users vector.

Parameters

- **mt_channel** (*np.ndarray*) – Channel of all users (2D numpy array).

- **desired_users** (*list[int]*) – An iterable with the indexes of the desired users or an integer.

Returns **mtSubmatrix** – Submatrix of the desired users (2D numpy array)

Return type np.ndarray

Notes

As an example, let's consider the case with a channel for 3 receivers, each with 2 receive antennas, where the transmitter has 6 transmit antennas.

```
>>> BD = BlockDiagonalizer(3, 0, 0)
>>> channel = np.vstack([np.ones([2, 6]), 2 * np.ones([2, 6]),
↪                 3 * np.ones([2, 6])])
>>> BD._get_sub_channel(channel, [0,2])
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [3., 3., 3., 3., 3., 3.],
       [3., 3., 3., 3., 3., 3.]])
>>> BD._get_sub_channel(channel, 0)
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

_get_tilde_channel(*mtChannel: numpy.ndarray*, *user: int*) → numpy.ndarray

Return the combined channel of all users except *user*.

Let k be the index for *user*. If the channel from all transmitters to receiver k is mtH_k, then this method returns $tilde{mtH_k} = [mtH_1^T, ldots, mtH_{k-1}^T, mtH_{k+1}^T, ldots, mtH_K]^T$.

Parameters

- **mtChannel** (*np.ndarray*) – Channel of all users (2D numpy array).
- **user** (*int*) – Index of the user.

Returns The combined channel of all users except *user*.

Return type np.ndarray

_perform_global_waterfilling_power_scaling(*Ms_bad: numpy.ndarray*, *Sigma: numpy.ndarray*) → numpy.ndarray

Perform the power scaling based on the water-filling algorithm for all the parallel channel gains in *Sigma*.

This power scaling method corresponds to maximizing the sum rate when the transmitter is a single base station transmitting to all users. Note that this approach may result in one or two "strong users" taking a dominant share of the available power.

Parameters

- **Ms_bad** (*np.ndarray*) – The previously calculated modulation matrix (without any power scaling). This should be a 2D numpy array.
- **Sigma** (*np.ndarray*) – The singular values of the effective channel when Ms_bad is applied. This should be a 1D numpy array of positive floats.

Returns **Ms_good** – The modulation matrix (2D numpy array) with the global power scaling applied.

Return type np.ndarray

_perform_normalized_waterfilling_power_scaling(*Ms_bad: numpy.ndarray*, *Sigma: numpy.ndarray*) → numpy.ndarray

Perform the power scaling based on the water-filling algorithm for all the parallel channel gains in *Sigma*, but normalize the result by the power of the base station transmitting with the highest power.

When we have a joint transmission where multiple base stations act as a single base station, then performing the water-filling on all the channels for the total available power may result in some base station transmitting with a higher power then it actually can. Therefore, we normalize the power of the strongest BS so that the power restriction at each BS is satisfied. This is sub-optimal since the other BSs will use less power then available, but it is simple and it works.

Parameters

- **Ms_bad** (*np.ndarray*) – The previously calculated modulation matrix (without any power scaling)
- **Sigma** (*np.ndarray*) – The singular values of the effective channel when Ms_bad is applied. This should be a 1D numpy array with positive values.

Returns **Ms_good** – The modulation matrix with the normalized power scaling applied. This is a 2D numpy array.

Return type np.ndarray

block_diagonalize(*mtChannel: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Perform the block diagonalization.

mtChannel is a matrix with the channel from the transmitter to all users, where each *iNUsers* rows correspond to one user.

For an example, see the documentation of the *BlockDiagonalizer* class.

Parameters **mtChannel** (*np.ndarray*) – Channel from (all) the transmitter(s) to all users. This should be a 2D numpy array.

Returns newH is a 2D numpy array corresponding to the Block diagonalized channel, while Ms_good is a 2D numpy array corresponding to the precoder matrix used to block diagonalize the channel.

Return type np.ndarray, np.ndarray

See also:

block_diagonalize_no_waterfilling()

block_diagonalize_no_waterfilling(*mtChannel: numpy.ndarray*) → Tuple[numpy.ndarray, numpy.ndarray]

Performs the block diagonalization, but without applying the water-filling algorithm.

The power of each base station is equally divided such that the square of the Frobenius norm or the columns of Ms_good corresponding to that base station is equal to its power.

Parameters **mtChannel** (*np.ndarray*) – Channel from (all) the transmitter(s) to all users. This should be a 2D numpy array.

Returns **(newH, Ms_good)** – newH is a 2D numpy array corresponding to the Block diagonalized channel, while Ms_good is a 2D numpy array corresponding to the precoder matrix used to block diagonalize the channel.

Return type (np.ndarray,np.ndarray)

See also:

block_diagonalize()

static calc_receive_filter(*newH: numpy.ndarray*) → numpy.ndarray
Calculates the Zero-Forcing receive filter.

Parameters **newH** (*np.ndarray*) – The block diagonalized channel (2D numpy array).

Returns **W_bd** – The zero-forcing matrix (2D numpy array) to separate each stream of each user.

Return type np.ndarray

class pyphysim.comm.blockdiagonalization.**EnhancedBD**(*num_users: int*, *iPu: float*, *noise_var: float*, *pe: float*)
Bases: *pyphysim.comm.blockdiagonalization.BDWithExtIntBase*

Class to perform the block diagonalization algorithm in a joint transmission scenario taking into account the external interference.

The EnhancedBD class performs the block diagonalization characteristic to the joint transmission scenario where multiple base stations act as a single transmitter to send data to the users. However, in addition to what the BlockDiagonalizer class does the EnhancedBD class can also take external interference into account.

One way to reduce of eliminate the external interference is to sacrifice streams in directions strongly occupied by the external interference.

Parameters

- **num_users** (*int*) – Number of users.
- **iPu** (*float*) – Power available for EACH user (in linear scale).
- **noise_var** (*float*) – Noise variance (power in linear scale).
- **pe** (*float*) – Power of the external interference source (in linear scale)

Notes

See the *BlockDiagonalizer* class for details about the block diagonalization process.

static _calc_linear_SINRs(*Heq_k_red: numpy.ndarray*, *Wk: numpy.ndarray*, *Re_k: numpy.ndarray*) → numpy.ndarray
Calculates the effective SINRs of each parallel channel.

Parameters

- **Heq_k_red** (*np.ndarray*) – Equivalent channel matrix of user *k* including the block diagonalization and any stream reduction applied (2D numpy array).
- **Wk** (*np.ndarray*) – Receive filter for user *k* (2D numpy array).
- **Re_k** (*np.ndarray*) – A numpy array where each element is the covariance matrix of the external interference PLUS noise seen by a user (1D numpy array of 2D numpy arrays.).

Returns **sinrs** – SINR (in linear scale) of all the parallel channels of all users.

Return type np.ndarray

_perform_BD_no_waterfilling_decide_number_streams(*mu_channel: pyphysim.channels.multiuser.MultiUserChannelMatrixE* → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]
Function called inside perform_BD_no_waterfilling when the stream reduction is performed and the number of sacrificed streams depend on the metric used (the function set as self._metric_func).

Parameters **mu_channel** (MultiUserChannelMatrixExtInt) – A MultiUserChannelMatrixExtInt object, which has the channel from all the transmitters to all the receivers, as well as th external interference.

Returns

- **MsPk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the precoder for a user (1D numpy array of 2D numpy arrays).
- **Wk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the receive filter for a user (1D numpy array of 2D numpy arrays).
- **Ns_all_users** (*np.ndarray*) – Number of streams of each user (1D numpy array of ints).

_perform_BD_no_waterfilling_fixed_or_naive_reduction(*mu_channel:* pyphysim.channels.multiuser.MultiUserChannelMat → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

Function called inside perform_BD_no_waterfilling when the naive or the fixed stream reduction should be performed.

For the naive or the fixed stream reduction cases the number of transmitted streams is always equal to self._metric_func_extra_args['num_streams']. That is, the number of sacrificed streams is equal to the number of transmit antennas minus num_streams.

The only difference between the naive and the fixed cases is that in the fixed case the reduction matrix P is chosen so that it gets as orthogonal to the external interference as possible, while the naive case simple chooses P as a submatrix of the diagonal matrix.

Parameters **mu_channel** (MultiUserChannelMatrixExtInt) – A MultiUserChannelMatrixExtInt object, which has the channel from all the transmitters to all the receivers, as well as th external interference.

Returns

- **MsPk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the precoder for a user (1D numpy array of 2D numpy arrays).
- **Wk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the receive filter for a user (1D numpy array of 2D numpy arrays).
- **Ns_all_users** (*np.ndarray*) – Number of streams of each user (1D numpy array of ints).

_perform_BD_no_waterfilling_no_stream_reduction(*mu_channel:* pyphysim.channels.multiuser.MultiUserChannelMatrixExtI → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

Function called inside perform_BD_no_waterfilling when no stream reduction should be performed.

Parameters **mu_channel** (MultiUserChannelMatrixExtInt) – A MultiUserChannelMatrixExtInt object, which has the channel from all the transmitters to all the receivers, as well as th external interference.

Returns

- **MsPk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the precoder for a user (1D numpy array of 2D numpy arrays).
- **Wk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the receive filter for a user (1D numpy array of 2D numpy arrays).
- **Ns_all_users** (*np.ndarray*) – Number of streams of each user (1D numpy array of ints).

block_diagonalize_no_waterfilling(*mu_channel:* pyphysim.channels.multiuser.MultiUserChannelMatrixExtInt) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

Perform the block diagonalization of *mu_channel* taking the external interference into account.

This is the main method calculating the BD algorithm. Two important parameters used here are the noise variance (an attribute of the *mu_channel* object) and the external interference power (the *pe* attribute) attributes.

Parameters **mu_channel** (*MultiUserChannelMatrixExtInt object.*) – A MultiUserChannelMatrixExtInt object, which has the channel from all the transmitters to all the receivers, as well as the external interference.

Returns

- **MsPk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the precoder for a user (1D numpy array of 2D numpy arrays).
- **Wk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the receive filter for a user (1D numpy array of 2D numpy arrays).
- **Ns_all_users** (*np.ndarray*) – Number of streams of each user (1D numpy array of ints).

static calc_receive_filter_user_k(*Heq_k_P:* numpy.ndarray, *P:* Optional[numpy.ndarray] = None) → numpy.ndarray

Calculates the Zero-Forcing receive filter of a single user *k* with or without the stream reduction.

Parameters

- **Heq_k_P** (*np.ndarray*) – The equivalent channel of user *k* after the block diagonalization process and any stream reduction (2D numpy array).
- **P** (*np.ndarray, optional*) – P has the most significant singular vectors of the external interference plus noise covariance matrix for each receiver.

Returns **W** – The receive filter of user *k* (2D numpy array).

Return type np.ndarray

Notes

If *P* is not None then the number of transmit streams will be equal to the number of columns in *P*. Also, the receive filter *W* includes a projection into the subspace spanned by the columns of *P*. Since *P* was calculated to be in the directions with weaker (or no) external interference then the receive filter *W* will mitigate the external interference.

property metric_name

Get name of the method used to decide how many streams to sacrifice.

Returns The metric name.

Return type str

set_ext_int_handling_metric(*metric:* Optional[str], *metric_func_extra_args_dict:* Optional[Dict[str, Any]] = None) → None

Set the metric used to decide how many streams to sacrifice for external interference handling.

The modification to the standard Block Diagonalization algorithm performed in this class consists in avoid transmit data in the subspace strongly occupied by the external interference source.

This sacrificing (not transmitting) of streams may or may not be worth it and different metrics can be used to decide this.

The valid values for the *metric* argument are 'None' (python None object), "fixed", "naive", "capacity" and "effective_throughput". Each of these values will impact on how the number of transmit streams is chosen and which subspace is actually used for the desired signal.

For the "fixed" and "naive" metrics, the number of transmit streams is determined by the value of the 'num_streams' key in the metric_func_extra_args_dict. The difference between them is how the subspace where the useful data is determined (for the given number of sacrificed streams).

For the "naive" metric, the stream reduction is performed by multiplying the usual block diagonalizing matrix M by a subset of the identity matrix. For the "fixed" metric the subspace containing the lowest remaining external interference energy is chosen by multiplying the block diagonalizing matrix M by the singular vectors of the external interference covariance matrix corresponding to the lowest singular values. The same procedure is used for the other metrics.

Differently from the "naive" and "fixed" metrics, the "capacity" and "effective_throughput" metrics try to determine this best number of sacrificed streams.

- If *metric* is None, then all streams will be used. That is, no streams will be sacrificed and the external interference won't be mitigated.
- If metric is "None" or "naive" then the specified number of streams will be used.
- If *metric* is 'capacity', then the metric used to decide how many streams to sacrifice will be the sum capacity. The function `calc_shannon_sum_capacity()` will be used to calculate the sum capacity metric, and since it only uses the SINR values, no extra arguments are required in the metric_func_extra_args_dict dictionary.
- If *metric* is 'effective_throughput' then the metric used to decide how many streams to sacrifice will be the effective throughput that can be obtained. The function *_calc_effective_throughput* will be used to calculate the effective throughput. Since it requires the a modulator and a packet length you should set the metric_func_extra_args_dict so that it has the keys 'modulator' and 'packet_length' with the correct values (a modulator object and an integer, respectively)

Parameters

- **metric** (*str | None*) – The metric name. Must be one of the available metrics: {None, 'capacity', 'effective_throughput'}.
- **metric_func_extra_args_dict** (*dict*) – A dictionary containing the extra arguments that must be passed to the metric function. For the "naive" and "fixed" metrics, this dictionary must contain the "num_streams" keyword with the desired number of transmit streams. For the "effective_throughput" metric this dictionary mst contain the "modulator" and "packet_length" keywords with a modulator object and an integer, respectively. For the other metrics metric_func_extra_args_dict will be ignored.

Raises **AttributeError** – If the metric is not one of the available metrics or if the metric_func_extra_args_dict does not contain the required keywords.

class pyphysim.comm.blockdiagonalization.**WhiteningBD**(*num_users: int*, *iPu: float*, *noise_var: float*, *pe: float*)

Bases: *pyphysim.comm.blockdiagonalization.BDWithExtIntBase*

Class to perform the block diagonalization algorithm in a joint transmission scenario taking into account the external interference.

Parameters

- **num_users** (*int*) – Number of users.
- **iPu** (*float*) – Power available for EACH user (in linear scale).
- **noise_var** (*float*) – Noise variance (power in linear scale).

- **pe** (*float*) – Power of the external interference source (in linear scale)

static _calc_receive_filter_with_whitening(*newH: numpy.ndarray*, *whitening_filter: numpy.ndarray*, *Nr: numpy.ndarray*, *Nt: numpy.ndarray*) → numpy.ndarray

Calculates the Zero-Forcing receive filter of all users.

Parameters

- **newH** (*np.ndarray*) – The block diagonalized channel (with the whitening applied). This should be a 2D numpy array.
- **whitening_filter** (*np.ndarray*) – The whitening filter of all users. This is a block diagonal matrix where each "block" is the whitening filter of a user.
- **Nr** (*np.ndarray*) – The number of receive antennas of each user.
- **Nt** (*np.ndarray*) – The number of transmit antennas of each user.

Returns **Wk_all_users** – A 1D numpy array where each element corresponds to the receive filter for a user. This is a 1D numpy array of 2D numpy arrays.

Return type np.ndarray

block_diagonalize_no_waterfilling(*mu_channel: pyphysim.channels.multiuser.MultiUserChannelMatrixExtInt*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

Perform the block diagonalization of *mu_channel* taking the external interference into account.

Parameters **mu_channel** (MultiUserChannelMatrixExtInt) – A MultiUserChannelMatrixExtInt object, which has the channel from all the transmitters to all the receivers, as well as the external interference.

Returns

- **Ms_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the precoder for a user (1D numpy array of 2D numpy arrays).
- **Wk_all_users** (*np.ndarray*) – A 1D numpy array where each element corresponds to the receive filter for a user (1D numpy array of 2D numpy arrays).
- **Ns_all_users** (*np.ndarray*) – Number of streams of each user (1D numpy array of ints).

pyphysim.comm.blockdiagonalization.**block_diagonalize**(*mtChannel: numpy.ndarray*, *num_users: int*, *iPu: float*, *noise_var: float*) → Tuple[numpy.ndarray, numpy.ndarray]

Performs the block diagonalization of *mtChannel*.

Parameters

- **mtChannel** (*np.ndarray*) – Global channel matrix (a 2D numpy array).
- **num_users** (*int*) – Number of users
- **iPu** (*float*) – Power available for each user
- **noise_var** (*float*) – Noise variance

Returns **(newH, Ms_good)** – newH is a 2D numpy array corresponding to the Block diagonalized channel, while Ms_good is a 2D numpy array corresponding to the precoder matrix used to block diagonalize the channel.

Return type (np.ndarray,np.ndarray)

Notes

The block diagonalization algorithm is described in [Spencer2004], where different power allocations are illustrated. The *BlockDiagonalizer* class implement two power allocation methods, a global power allocation, and a 'per transmitter' power allocation.

pyphysim.comm.blockdiagonalization.**calc_receive_filter**(*newH: numpy.ndarray*) → numpy.ndarray

Calculates the Zero-Forcing receive filter.

Parameters **newH** (*np.ndarray*) – The block diagonalized channel.

Returns The zero-forcing matrix to separate each stream of each user. Note that W_bd_H is directly applied to the received signals (no need to calculate the conjugate transpose).

Return type np.ndarray

pyphysim.comm.waterfilling module

Implements a waterfilling method.

The doWF method performs the waterfilling algorithm.

pyphysim.comm.waterfilling.**doWF**(*vtChannels: numpy.ndarray*, *dPt: float*, *noiseVar: float = 1.0*, *Es: float = 1.0*) → Tuple[numpy.ndarray, float]

Performs the Waterfilling algorithm and returns the optimum power and water level.

Parameters

- **vtChannels** (*np.ndarray*) – Numpy array with the channel POWER gains (power of the parallel AWGN channels).
- **dPt** (*float*) – Total available power.
- **noiseVar** (*float*) – Noise variance (power in linear scale).
- **Es** (*float*) – Symbol energy (in linear scale).

Returns **(vtOptP, mu)** – A tuple with vtOptP and mu, where vtOptP are the optimum powers, while mu is the water level.

Return type (np.ndarray, float)

Module contents

Package with communication related modules such as modulators, channels, etc.

1. summary
2. extended summary
3. routine listings
4. see also
5. notes
6. references
7. examples

pyphysim.extra package

Subpackages

pyphysim.extra.MATLAB package

Submodules

pyphysim.extra.MATLAB.python2MATLAB module

Module with functions to easily moving data from python to MATLAB.

pyphysim.extra.MATLAB.python2MATLAB.**to_mat_str**(*x*, *format_string='+.12e'*)

Convert the ndarray 'x' to a string corresponding to the MATLAB representation of *x*.

The to_mat_str function formats numpy arrays of arbitrary dimension in a way which can easily copied and pasted into an interactive MATLAB session

Parameters

- **x** (*numpy array*) – The numpy array to be represented as a MATLAB type.
- **format_string** (*str, optional*) – The format_string string to convert each element in *x*.

Returns **converted_string** – A string that represents the converted numpy array. You can copy this string and past it into a MATLAB session.

Return type str

Examples

```
>>> a=np.arange(1,10)
>>> a.shape=(3,3)
>>> # Print as a numpy matrix
>>> print(a)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> # Call to_mat_str(a) to print the string representation of the
>>> # converted matrix
>>> print(to_mat_str(a))
[+1.000000000000e+00, +2.000000000000e+00, +3.000000000000e+00; +4.
↪000000000000e+00, +5.000000000000e+00, +6.000000000000e+00; +7.000000000000e+00,
↪ +8.000000000000e+00, +9.000000000000e+00]
```

Module contents

Package with MATLAB related functions.

Examples: conversions for the print output from python to MATLAB and vice-verse. That way you can print a numpy array, copy the output and past in MATLAB for further analysis.

Submodules

pyphysim.extra.pgfplotshelper module

This module contains useful functions to create pgfplots code (latex code using the pgfplots package) from python data.

One example of tex code for a plot using pgfplots is show below

```
\begin{tikzpicture}
  \begin{axis}[axis options]
    \addplot [plot options]
    plot [options]
    coordinates {
      (0, 0)
      (1, 1)
      (2, 4)
      (3, 9)};
    \addlegendentry{Legend of the last line};
  \end{axis}
\end{tikzpicture}
```

pyphysim.extra.pgfplotshelper.**generate_pgfplots_plotline**(*x*, *y*, *errors=None*, *options=None*, *legend=None*)

This function generates the code corresponding to the "addplot" command in a pgfplots plot for the coordinates given in *x* and *y*.

If the parameter *errors* is also provided then error bars will be added in the *y* direction, while options to the addplot command can be passed as a string in the *options* argument.

Parameters

- **x** (*np.ndarray | list[float] | list[int]*) – The data for the 'x' axis in the plot.
- **y** (*np.ndarray | list[float] | list[int]*) – The data for the 'x' axis in the plot
- **errors** (*np.ndarray | list[float] | list[int], optional*) – The error for plotting the errorbars.
- **options** (*str*) – pgfplot options for the plot line. Ex: "color=red, solid, mark=square, mark options={solid}"
- **legend** (*str*) – The legend for the plot line.

Module contents

Package containing extra functionality.

pyphysim.ia package

Submodules

pyphysim.ia.algorithms module

Module with implementation of Interference Alignment (IA) algorithms.

Note that all IA algorithms require the channel object and any change to the channel object must be performed before calling the *solve* method of the IA algorithm object. This includes generating the channel and setting the noise variance.

class pyphysim.ia.algorithms.**AlternatingMinIASolver**(*multiUserChannel:* pyphysim.channels.multiuser.MultiUserChannelMatrix)

Bases: *pyphysim.ia.algorithms.IterativeIASolverBaseClass*

Implements the "Interference Alignment via Alternating Minimization" algorithm from the paper with the same name [PetersHeathAltMin2009].

This algorithm is applicable to a "K-user" scenario and it is very flexible in the sense that you can change the number of transmit antennas, receive antennas and streams per user, as well as the number of users involved in the IA process. However, note that alignment is only feasible for some cases configurations.

An example of a common scenario is a scenario with 3 pairs or transmitter/receiver with 2 antennas in each node and 1 stream transmitted per node.

You can determine the scenario of an AlternatingMinIASolver object by inferring the variables K, Nt, Nr and Ns.

Parameters **multiUserChannel** (*muchannels.MultiUserChannelMatrix*) – The multiuser channel.

Notes

_before_initialize_W_func() → None
Method run in any of the initialize methods after the precoder is initialized but before the receive filter is initialized.

_solve_finalize() → None
Perform any post processing after the solution has been found.

_step() → None
Performs one iteration of the algorithm.

The step method is usually all you need to call to perform an iteration of the Alternating Minimization algorithm. It will update C, then update F and finally update W.

See also:

_updateC(), *_updateF()*, *_updateW()*

_updateC() → None
Update the value of Ck for all K users.

Ck contains the orthogonal basis of the interference subspace of user k. It corresponds to the Nk-Sk dominant eigenvectors of

$\sum_{l \neq k} {}_{k,l} l {}^{H}_{l} {}^{H}_{k,l}$.

Notes

This method is called in the *_step()* method.

See also:

_step()

_updateF() → None
Update the value of the precoder of all K users.

Fl, the precoder of the l-th user, tries avoid as much as possible to send energy into the desired signal subspace of the other users. Fl contains the Sl least dominant eigenvectors of $\sum_{k \neq l} H^{H}_{k,l}(I - C_k C^{H}_{k}) H_{k,l}$

Notes

This method is called in the *_step()* method.

See also:

_step()

_updateW() → None
Update the zero-forcing filters.

The zero-forcing filter is calculated in the paper "MIMO Interference Alignment Over Correlated Channels with Imperfect CSI".

Notes

This method is called in the *_step()* method.

See also:

_step()

get_cost() → float
Get the Cost of the algorithm for the current iteration of the precoder.

Returns **cost** – The Cost of the algorithm for the current iteration of the precoder. This is a real non-negative number.

Return type float

property initialize_with
Get method for the initialize_with property.

class pyphysim.ia.algorithms.**BruteForceStreamIASolver**(*iasolver_obj:* pyphysim.ia.algorithms.IterativeIASolverBaseClass)

Bases: object

Implements the Brute Force Stream Interference Alignment algorithm variation.

This is not a new IA algorithm, but rather a variation of existing IA algorithms. The idea is to use another IA algorithm to find the IA solution for each possible stream configuration (number of streams for each user) and keep the best one.

Note: IA algorithms can provide different performance for the same number of streams but with different initializations. To reduce this variability we set the initialization method to 'svd' (see 'initialize_with' property in the IterativeIASolverBaseClass class) so that the initialization is always the same.

Parameters **iasolver_obj** (*T <= IASolverBaseClass*) – Must be an object of a derived class of IterativeIASolverBaseClass.

clear() → None
Clear the BruteForceStreamIASolver object.

property every_sum_capacity
Get method for the every_sum_capacity property.

Returns Tuple containing the sum capacity for each stream combination in *self.stream_combinations*.

Return type list

property runned_iterations
Get method for the runned_iterations property.

Returns The number of runned iterations.

Return type int

solve(*Ns: IntOrIntSequence*, *P: Optional[FloatOrFloatSequence] = None*) → int
Find the IA solution.

This method updates the 'F' and 'W' member variables.

Parameters

- **Ns** (*int | np.ndarray*) – MAXIMUM number of streams of each user. All possible values from 1 to Ns (or Ns[k], if Ns is an array) will tried.
- **P** (*np.ndarray | List[float] | float, optional*) – Power of each user. If not provided, a value of 1 will be used for each user.

Returns Number of iterations the iterative interference alignment algorithm run.

Return type int

property stream_combinations
Get method for the stream_combinations property.

Returns Tuple containing every possible stream combination.

Return type tuple

class pyphysim.ia.algorithms.**ClosedFormIASolver**(*multiUserChannel: pyphysim.channels.multiuser.MultiUserChannelMatrix*, *use_best_init: bool = True*)
Bases: *pyphysim.ia.iabase.IASolverBaseClass*

Implements the closed form Interference Alignment algorithm as described in the paper "Interference Alignment and Degrees of Freedom of the K User Interference Channel [CadambeDoF2008]".

Parameters

- **multiUserChannel** (*muchannels.MultiUserChannelMatrix*) – The multiuser channel.
- **use_best_init** (*bool*) – If true, all possible initializations (subsets of the eigenvectors of the 'E' matrix)for the first precoder will be tested and the best solution will be used. If false, the first initialization will be used.

Notes

_calc_E() → numpy.ndarray
Calculates the "E" matrix, given by

$=_{31}^{-1}{}_{32}{}_{12}^{-1}{}_{13}{}_{23}^{-1}{}_{21}$.

Returns The "E" matrix.

Return type np.ndarray

_calc_all_F_initializations(*Ns: int*) → List[numpy.ndarray]
Calculates all possible initializations for the first precoder (self._F[0]).

The precoder self._F[0] is initialized with a subset of the eigenvectors of the matrix 'E'. Therefore, this method returns all possible subsets of the eigenvectors of the matrix 'E'.

Parameters **Ns** (*int*) – Number of streams of the first user.

Returns **all_initializations** – All possible subsets (with size Ns) of the eigenvectors of the matrix E

Return type list[np.ndarray]

_updateF(*F0: Optional[numpy.ndarray] = None*) → None
Find the precoders.

Parameters **F0** (*np.ndarray*) – The first precoder. If not provided, the matrix 'E' will be calculated (with the _calc_E method) and the the first Ns eigenvectors will be used as F0.

_updateW() → None
Find the receive filters

solve(*Ns: IntOrIntSequence*, *P: Optional[FloatOrFloatSequence] = None*) → None
Find the IA solution.

This method updates the 'F' and 'W' member variables.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray | List[float] | float, optional*) – Power of each user. If not provided, a value of 1 will be used for each user.

class pyphysim.ia.algorithms.**GreedStreamIASolver**(*iasolver_obj: Type[*pyphysim.ia.iabase.IASolverBaseClass*]*)

Bases: object

Implements the Greed Stream Interference Alignment algorithm variation.

This is not a new IA algorithm, but rather a variation of existing IA algorithms. The idea is to use another IA algorithm to find the IA solution for the desired maximum number of streams. After the solution is found, we remove the worst stream and use the same algorithm again to find a new solution. If the solution after the stream reduction provided a larger sum capacity we remove the worst stream again and keep going until each user has only one stream or the sum capacity after stream reduction is lower. The final solution will be for the number of streams that yielded the largest sum capacity.

Parameters **iasolver_obj** (*T <= IASolverBaseClass*) – Must be an object of a derived class of IterativeIASolverBaseClass.

_find_index_stream_with_worst_sinr() → Tuple[int, int]
Considering the current solution (precoders and receive filters) in self._iasolver, find the index of the user and stream corresponding to the worst SINR.

Returns

- **user_idx** (*int*) – The index of the user that has the stream with the worst SINR.
- **stream_idx** (*int*) – The index of the stream (for user *user_idx*) with the worst SINR.

property runned_iterations
Get method for the runned_iterations property.

Returns The number of runned iterations.

Return type int

solve(*Ns: IntOrIntSequence*, *P: Optional[FloatOrFloatSequence] = None*) → int
Find the IA solution.

This method updates the 'F' and 'W' member variables.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray | List[float] | float, optional*) – Power of each user. If not provided, a value of 1 will be used for each user.

Returns Number of iterations the iterative interference alignment algorithm run.

Return type int

class pyphysim.ia.algorithms.**IterativeIASolverBaseClass**(*multiUserChannel:* pyphysim.channels.multiuser.MultiUserChannelMatri

Bases: *pyphysim.ia.iabase.IASolverBaseClass*

Base class for all Iterative IA algorithms.

All subclasses of IterativeIASolverBaseClass must implement at least the _updateF and _updateW methods.

Solving an iterative algorithm usually involves some initialization and then performing a "step" a given number of times until convergence. The initialization code is performed in the *_solve_init* method while the "step" corresponds to the *_step* method.

The initialization code is defined here as simply initializing the precoder with a random matrix and then calling the *_updateW* method (which must be implemented in a subclass) to update the receive filter. This is usually what you want but any subclass can redefine *_solve_init* if a different initialization is required.

The "step" part usually involves updating the precoder and then updating the receive filters. The definition of *_step* here calls two methods that MUST be defined in subclasses, the _updateF method and the _updateW method. If anything else is required in the "step" part then the _step method can be redefined in a subclass, but even in that case it should call the _updateF and _updateW methods instead of implementing everything in your redefined _step method.

Parameters **multiUserChannel** (*muchannels.MultiUserChannelMatrix*) – The multiuser channel.

_before_initialize_W_func() → None
Method run in any of the initialize methods after the precoder is initialized but before the receive filter is initialized.

_dont_initialize_F_and_only_and_find_W(**_: Any*) → None
Initialize the IA Solution from a random matrix.

The implementation here simple initializes the precoder variable and then calculates the initial receive filter.

Note: The *dummy1* and *dummy2* arguments have no effect. They only exist to keep the signature of this method equal to the signature of other *initialize* methods.

_initialize_F_and_W_from_alt_min(*Ns: IntOrIntSequence*, *P: numpy.ndarray*) → None
Initialize the IA Solution from the Alternating Minimizations IA solver.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray*) – Power of each user. If not provided, a value of 1 will be used for each user.

_initialize_F_and_W_from_closed_form(*Ns: IntOrIntSequence*, *P: numpy.ndarray*) → None
Initialize the IA Solution from the closed form IA solver.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray*) – Power of each user. If not provided, a value of 1 will be used for each user.

_initialize_F_randomly_and_find_W(*Ns: Sequence[int]*, *P: numpy.ndarray*) → None
Initialize the IA Solution from a random matrix.

The implementation here simple initializes the precoder variable and then calculates the initial receive filter.

Parameters

- **Ns** (*int np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray*) – Power of each user. If not provided, a value of 1 will be used for each user.

_initialize_F_with_svd_and_find_W(*Ns: IntOrIntSequence*, *P: numpy.ndarray*) → None
Initialize the IA Solution from the most significant singular vectors of each user's channel.

The implementation here simple initializes the precoder variable of each user as the most significant singular vector(s) of that user. After that, calculate the initial receive filters.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray*) – Power of each user. If not provided, a value of 1 will be used for each user.

classmethod _is_diff_significant(*F_old: numpy.ndarray*, *F_new: numpy.ndarray*, *relative_factor: float*) → bool
Test if there was any significant change from *F_old* to *F_new*.

This method is used internally in the solve method of the IterativeIASolverBaseClass to detect when the precoder of a given iteration didn't change significantly from one iteration to another. This is used to stop the iterations of the algorithm and avoid unnecessary computations.

Parameters

- **F_old** (*np.ndarray*) – The precoder of all users (in a previous iteration). This is a 1D numpy array of numpy arrays.
- **F_new** (*np.ndarray*) – The precoder of all users (in the current iteration). This is a 1D numpy array of numpy arrays.
- **relative_factor** (*float*) – Relative change of the precoder in one iteration to the next one. If the relative change from one iteration to the next one is lower than this factor then the algorithm will stop the iterations before the max_iterations limit is reached.

Returns **out** – True if the difference is significant, False otherwise.

Return type bool

Notes

A difference is considered significant if it is larger then 1/1000 of the minimum value in the precoder.

_solve_finalize() → None
Perform any post processing after the solution has been found.

Some of the found precoders may be a singular matrix. In that case, we need to remove the dimensions with zero energy from both the found precoder and the receive filter.

_solve_init(*Ns: IntOrIntSequence*, *P: FloatOrFloatSequence*) → None
Code run in the *solve* method before the loop that run the *_step()* method.

The implementation here simple initializes the precoder variable and then calculates the initial receive filter.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*float | np.ndarray*) – Power of each user. If not provided, a value of 1 will be used for each user.

_step() → None
Performs one iteration of the algorithm.

This method does not return anything, but instead updates the precoder and receive filter.

abstract _updateF() → None
Update the precoders.

Notes

This method should be implemented in the derived classes

See also:

_step()

abstract _updateW() → None
Update the receive filters.

Notes

This method should be implemented in the derived classes

See also:

_step()

clear() → None
Clear the IA Solver object.

All member attributes that are updated during the solve method, such as the precoder and receive filters, will be cleared. The other attributes that correspond to "configuration" such as the channel object won't be changed

Notes

You should overwrite this method in subclasses that pass parameters to the __init__ method, since here we call __init__ without arguments which is probably not what you want.

property initialize_with
Get method for the initialize_with property.

randomizeF(*Ns: Union[int, List[int], Sequence[int]]*, *P: Optional[numpy.ndarray] = None*) → None
Generates a random precoder for each user.

Parameters

- **Ns** (*int | list[int] | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray, optional*) – Power of each user. If not provided, a value of 1 will be used for each user.

property runned_iterations
Get method for the runned_iterations property.

solve(*Ns: IntOrIntSequence*, *P: Optional[FloatOrFloatSequence] = None*) → int
Find the IA solution by performing the *_step()* method several times.

The number of times the *_step()* method is run is controlled by the max_iterations member variable.

Before calling the *_step()* method for the first time the *_solve_init* method is called to perform any required initializations. Since iterative IA algorithms usually starts with a random precoder then the *_solve_init* implementation in IterativeIASolverBaseClass calls randomizeF.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray | List[float] | float, optional*) – Power of each user. If not provided, a value of 1 will be used for each user.

Returns

- *Number of iterations that the iterative interference alignment*
- *algorithm run.*

Notes

You probably should not overwrite this method in sub-classes of the IterativeIASolverBaseClass. If you want to change the initialization of the algorithm overwrite the _solve_init method instead.

Subclasses must implement the _updateF() and _updateW() methods. If something else besides calling these two methods is required in the "step" part of the algorithm, then reimplement also the _step() method.

class pyphysim.ia.algorithms.**MMSEIASolver**(*multiUserChannel:* pyphysim.channels.multiuser.MultiUserChannelMatrix)
Bases: *pyphysim.ia.algorithms.IterativeIASolverBaseClass*

Implements the MMSE based Interference Alignment algorithm.

This algorithm is applicable to a "K-user" scenario and it is described in [Peters2011].

An example of a common scenario is a scenario with 3 pairs or transmitter/receiver with 2 antennas in each node and 1 stream transmitted per node.

You can determine the scenario of an MMSEIASolver object by inferring the variables K, Nt, Nr and Ns.

Parameters **multiUserChannel** (*muchannels.MultiUserChannelMatrix*) – The multiuser channel.

Notes

_calc_Uk(*k: int*) → numpy.ndarray

Calculates the receive filter of the k-th user.

Parameters **k** (*int*) – User index

Returns **Uk** – The receive filter of the user 'k'.

Return type np.ndarray

_calc_Vi(*i: int*, *mu_i: Optional[float] = None*) → numpy.ndarray

Calculates the precoder of the i-th user.

Parameters

- **i** (*int*) – User index
- **mu_i** (*float, optional*) – The value of the Lagrange multiplier. If it is None (default), then the best value will be found and used to calculate the precoder.

Returns **Vi** – The calculate precoder of the i-th user.

Return type np.ndarray

static _calc_Vi_for_a_given_mu(*sum_term: numpy.ndarray*, *mu_i: float*, *H_herm_U: numpy.ndarray*) → numpy.ndarray

Calculates the value of Vi for the given parameters.

This method is called inside _calc_Vi.

Parameters

- **sum_term** (*np.ndarray*) – The summation term in the formula to calculate the precoder.
- **mu_i** (*float*) – The value of the lagrange multiplier
- **H_herm_U** (*np.ndarray*) – The value of $H_i i^H U_i$

Returns The Vi matrix for the given parameters.

Return type np.ndarray

static _calc_Vi_for_a_given_mu2(*inv_sum_term: numpy.ndarray*, *mu_i: float*, *H_herm_U: numpy.ndarray*) → numpy.ndarray

Calculates the value of Vi for the given parameters.

This method is called inside _calc_Vi.

Parameters

- **inv_sum_term** (*np.ndarray*) – The inverse of the summation term in the formula to calculate the precoder when mu_i is equal to zero.
- **mu_i** (*float*) – The value of the lagrange multiplier
- **H_herm_U** (*np.ndarray*) – The value of $H_i i^H U_i$

Returns The Vi matrix for the given parameters.

Return type np.ndarray

_solve_init(*Ns: IntOrIntSequence*, *P: numpy.ndarray*) → None
Code run in the *solve* method before the loop that run the *IterativeIASolverBaseClass._step()* method.

The implementation here simple initializes the precoder variable and then calculates the initial receive filter.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray*) – Power of each user. If not provided, a value of 1 will be used for each user.

_updateF() → None
Updates the precoder of all users.

_updateW() → None
Updates the receive filter of all users.

class pyphysim.ia.algorithms.**MaxSinrIASolver**(*multiUserChannel:* pyphysim.channels.multiuser.MultiUserChannelMatrix)
Bases: *pyphysim.ia.algorithms.IterativeIASolverBaseClass*

Implements the "Interference Alignment via Max SINR" algorithm.

This algorithm is applicable to a "K-user" scenario and it is described in [Cadambe2008].

An example of a common scenario is a scenario with 3 pairs or transmitter/receiver with 2 antennas in each node and 1 stream transmitted per node.

You can determine the scenario of an MaxSinrIASolver object by inferring the variables K, Nt, Nr and Ns.

Parameters **multiUserChannel** (*muchannels.MultiUserChannelMatrix*) – The multiuser channel.

_calc_Bkl_cov_matrix_all_l_rev(*k: int*) → numpy.ndarray
Calculates the interference-plus-noise covariance matrix for all streams at "receiver" k for the reverse channel.

Parameters **k** (*int*) – Index of the desired user.

Returns **Bkl_rev** – Covariance matrix of all streams of user k. Each element of the returned 1D numpy array is a 2D numpy complex array corresponding to the covariance matrix of one stream of user k.

Return type np.ndarray

_calc_Bkl_cov_matrix_first_part_rev(*k: int*) → numpy.ndarray
Calculates the first part in the equation of the Blk covariance matrix of the reverse channel.

Parameters **k** (*int*) – Index of the desired user.

Returns **Bkl_first_part_rev** – First part in equation (28) of [Cadambe2008], but for the reverse channel.

Return type np.ndarray

See also:

_calc_Bkl_cov_matrix_first_part_rev()

_calc_Bkl_cov_matrix_second_part_rev(*k: int*, *l: int*) → numpy.ndarray
Calculates the second part in the equation of the Blk covariance matrix of the reverse channel..

The second part is given by

$\frac{P^{[k]}}{d^{[k]}} {}^{[kk]}_{\star l} {}^{[k]}_{\star l} [k]\dagger [kk]\dagger$

Parameters

- **k** (*int*) – Index of the desired user.
- **l** (*int*) – Index of the desired stream.

Returns **second_part** – Second part in equation (28) of [Cadambe2008].

Return type np.ndarray

classmethod _calc_Uk(*Hkk: numpy.ndarray*, *Vk: numpy.ndarray*, *Bkl_all_l: numpy.ndarray*) → numpy.ndarray

Similar to the *_calc_Ukl()* method, but while *_calc_Ukl()* calculates the receive filter (a vector) only for the l-th stream *_calc_Uk()* calculates a receive filter (a matrix) for all streams.

Parameters

- **Hkk** (*np.ndarray*) – Channel from transmitter K to receiver K.
- **Vk** (*np.ndarray*) – Precoder of user k.
- **Bkl_all_l** (*np.ndarray*) – Covariance matrix of all streams of user k. Each element of the returned 1D numpy array is a 2D numpy complex array corresponding to the covariance matrix of one stream of user k.

Returns **Uk** – The receive filter for all streams of user k.

Return type np.ndarray

_calc_Uk_all_k() → numpy.ndarray

Calculates the receive filter of all users.

Returns The receive filter of all users. This is a numpy array of numpy arrays.

Return type np.ndarray

_calc_Uk_all_k_rev() → numpy.ndarray

Calculates the receive filter of all users for the reverse channel.

Returns

np.ndarray The receive filter of all users for the reverse channel. This is a numpy array of numpy arrays.

classmethod _calc_Ukl(*Hkk: numpy.ndarray*, *Vk: numpy.ndarray*, *Bkl: numpy.ndarray*, *l: int*) → numpy.ndarray

Calculates the Ukl matrix in equation (29) of [Cadambe2008].

Parameters

- **Hkk** (*np.ndarray*) – Channel from transmitter K to receiver K.
- **Vk** (*np.ndarray*) – Precoder of user k.
- **Bkl** (*np.ndarray*) – The previously calculates Bkl matrix in equation (28) of [Cadambe2008]
- **l** (*int*) – Index of the desired stream

Returns **Ukl** – The calculated Ukl matrix. This is a 2D numpy array (with self.Nr[k] rows and a single column).

Return type np.ndarray

_updateF() → None
Update the precoders.

Notes

This method is called in the *IterativeIASolverBaseClass._step()* method.

See also:

IterativeIASolverBaseClass._step()

_updateW() → None
Update the receive filters.

Notes

This method is called in the *IterativeIASolverBaseClass._step()* method.

See also:

IterativeIASolverBaseClass._step()

class pyphysim.ia.algorithms.**MinLeakageIASolver**(*multiUserChannel:* pyphysim.channels.multiuser.MultiUserChannelMatrix)
Bases: *pyphysim.ia.algorithms.IterativeIASolverBaseClass*

Implements the Minimum Leakage Interference Alignment algorithm as described in the paper "Approaching the Capacity of Wireless Networks through Distributed Interference Alignment [Cadambe2008]".

Parameters **multiUserChannel** (*muchannels.MultiUserChannelMatrix*) – The multiuser channel.

_calc_Uk_all_k() → numpy.ndarray
Calculates the receive filter of all users.

Returns The Uk array of each user. This is a 1D numpy array of numpy arrays.

Return type np.ndarray

_calc_Uk_all_k_rev() → numpy.ndarray
Calculates the receive filter of all users in the reverse network.

Returns The Uk array of each user. This is a 1D numpy array of numpy arrays.

Return type np.ndarray

_updateF() → None
Update the precoders.

Notes

This method is called in the *IterativeIASolverBaseClass._step()* method.

See also:

IterativeIASolverBaseClass._step()

_updateW() → None
Update the receive filters.

Notes

This method is called in the *IterativeIASolverBaseClass._step()* method.

See also:

IterativeIASolverBaseClass._step()

get_cost() → float
Get the Cost of the algorithm for the current iteration of the precoder.

For the Minimum Leakage Interference Alignment algorithm the cost is equivalent to the sum of the interference that all users see after applying the receive filter. That is,

$$C = Tr[_k^H{}_{kk}]$$

Returns **cost** – The Cost of the algorithm for the current iteration of the precoder. This is a (real non-negative number).

Return type float

pyphysim.ia.iabase module

Module containing the base class for Interference Alignment (IA) Algorithms.

This module should probably only be imported in the other modules inside the 'ia' package that implement the IA algorithms.

class pyphysim.ia.iabase.**IASolverBaseClass**(*multiUserChannel:* pyphysim.channels.multiuser.MultiUserChannelMatrix)

Bases: object

Base class for all Interference Alignment Algorithms.

At least the *_updateW*, *_updateF* and *solve* methods must be implemented in the subclasses of IASolverBaseClass, where the *solve* method uses the *_updateW* and *_updateF* methods in its implementation.

The implementation of the *_updateW* method should call the *_clear_receive_filter* method in the beginning and after that set either the _W or the _W_H variables with the correct value.

The implementation of the *_updateF* method should call the clear_precoder_filter in the beginning and after that set _W variable with the correct precoder (normalized to have a Frobenius norm equal to one.

The implementation of the *_updateF* method must set the _F variable with the correct value.

Another method that can be implemented is the get_cost method. It should return the cost of the current IA solution. What is considered "the cost" varies from one IA algorithm to another, but should always be a real non-negative number. If get_cost is not implemented a value of -1 is returned.

Parameters **multiUserChannel** (*muchannels.MultiUserChannelMatrix*) – The multiuser channel.

property F
Transmit precoder of all users.

Returns The precoders of all users (a 1D numpy array of 2D numpy arrays).

Return type np.ndarray

property K
The number of users.

Returns **K** – The number of users.

Return type int

property Nr
Number of receive antennas of all users.

Returns **Nr** – Number of receive antennas of all users.

Return type np.ndarray

property Ns
Number of streams of all users.

Returns **Ns** – Number of streams of all users.

Return type np.ndarray

property Nt
Number of transmit antennas of all users.

Returns **Nt** – Number of transmit antennas of all users.

Return type np.ndarray

property P
Transmit power of all users.

Returns The power of all users.

Return type np.ndarray

property W
Receive filter of all users.

Returns The receive filter of all users. (a 1D numpy array of 2D numpy arrays).

Return type np.ndarray

property W_H
Get method for the W_H property.

Returns The conjugate of the receive filter of all users. (a 1D numpy array of 2D numpy arrays).

Return type np.ndarray

_calc_Bkl_cov_matrix_all_l(*k: int*, *noise_power: Optional[float] = None*) → numpy.ndarray
Calculates the interference-plus-noise covariance matrix for all streams at receiver k according to equation (28) in [Cadambe2008].

The interference-plus-noise covariance matrix for stream l of user k is given by Equation (28) in [Cadambe2008], which is reproduced below

$$ {}^{[kl]} = \sum_{j=1}^{K} \frac{P^{[j]}}{d^{[j]}} \sum_{d=1}^{d^{[j]}} {}^{[kj]}_{\star l} {}^{[j]}_{\star l} {}^{[j]\dagger} {}^{[kj]\dagger} - \frac{P^{[k]}}{d^{[k]}} {}^{[kk]}_{\star l} {}^{[k]}_{\star l} {}^{[k]\dagger} {}^{[kk]\dagger} + {}_{N^{[k]}} $$

where $P^{[k]}$ is the transmit power of transmitter k, $d^{[k]}$ is the number of degrees of freedom of user k, ${}^{[kj]}$ is the channel between transmitter j and receiver k, ${}_{\star l}$ is the l-th column of the precoder of user k and ${}_{N^k}$ is an identity matrix with size equal to the number of receive antennas of receiver k.

Parameters

- **k** (*int*) – Index of the desired user.
- **noise_power** (*float*) – Noise power (variance).

Returns **Bkl** – Covariance matrix of all streams of user k. Each element of the returned 1D numpy array is a 2D numpy complex array corresponding to the covariance matrix of one stream of user k.

Return type np.ndarray

Notes

To be simple, a function that returns the covariance matrix of only a single stream "l" of the desired user "k" could be implemented, but in the order to calculate the max SINR algorithm we need the covariance matrix of all streams and returning them in single function as is done here allows us to calculate the first part in equation (28) of [Cadambe2008] only once, since it is the same for all streams.

_calc_Bkl_cov_matrix_first_part (*k: int*) → numpy.ndarray
Calculates the first part in the equation of the Blk covariance matrix in equation (28) of [Cadambe2008].

The first part is given by

$$\sum_{j=1}^{K} \frac{P^{[j]}}{d^{[j]}} \sum_{d=1}^{d^{[j]}} {}^{[kj]}_{\star d}\, {}^{[j]}_{\star d}\, {}^{[j]\dagger}\, {}^{[kj]\dagger}$$

Note that it only depends on the value of k.

Parameters **k** (*int*) – Index of the desired user.

Returns **Bkl_first_part** – First part in equation (28) of [Cadambe2008].

Return type np.ndarray

_calc_Bkl_cov_matrix_second_part (*k: int*, *l: int*) → numpy.ndarray
Calculates the second part in the equation of the Blk covariance matrix in equation (28) of [Cadambe2008] (note that it does not include the identity matrix).

The second part is given by

$$\frac{P^{[k]}}{d^{[k]}} \, {}^{[kk]}_{\star l} \, {}^{[k]}_{\star l} \, {}^{[k]\dagger} \, {}^{[kk]\dagger}$$

Parameters

- **k** (*int*) – Index of the desired user.
- **l** (*int*) – Index of the desired stream.

Returns **second_part** – Second part in equation (28) of [Cadambe2008].

Return type np.ndarray

_calc_SINR_k (*k: int*, *Bkl_all_l: Sequence[numpy.ndarray]*) → numpy.ndarray
Calculates the SINR of all streams of user 'k'.

Parameters

- **k** (*int*) – Index of the desired user.
- **Bkl_all_l** (*list[np.ndarray] | nd.ndarray*) – A sequence (1D numpy array, a list, etc) of 2D numpy arrays corresponding to the Bkl matrices for all 'l's.

Returns **SINR_k** – The SINR for the different streams of user k.

Return type np.ndarray

_calc_equivalent_channel(*k: int*) → numpy.ndarray
Calculates the equivalent channel for user k considering the effect of the precoder (including transmit power), the actual channel, and the receive filter (without power compensation).

Parameters **k** (*int*) – The index of the desired user.

Returns **Hk_eq** – The equivalent channel.

Return type np.ndarray

Notes

This method is used only internally in order to calculate the "W" get property so that the returned filter W compensates the effect of the direct channel.

_clear_precoder_filter() → None
Clear the precoder filter.

This should be called in the beginning of the implementation of the *_updateF* method in subclasses.

_clear_receive_filter() → None
Clear the receive filter.

This should be called in the beginning of the implementation of the *_updateW* method in subclasses.

_get_channel(*k: int*, *l: int*) → numpy.ndarray
Get the channel from transmitter l to receiver k.

Parameters

- **l** (*int*) – Transmitting user.
- **k** (*int*) – Receiving user.

Returns **H** – The channel matrix between transmitter l and receiver k.

Return type np.ndarray

_get_channel_rev(*k: int*, *l: int*) → numpy.ndarray
Get the channel from transmitter l to receiver k in the reverse network.

Let the matrix $_{kl}$ be the channel matrix between the transmitter l to receiver k in the direct network. The channel matrix between the transmitter l to receiver k in the reverse network, denoted as $\leftarrow_{kl}$, is then given by $\leftarrow_{kl} =^{\dagger}_{lk}$ where † is the conjugate transpose of .

Parameters

- **l** (*int*) – Transmitting user of the reverse network.
- **k** (*int*) – Receiving user of the reverse network.

Returns **H** – The channel matrix between transmitter l and receiver k in the reverse network.

Return type np.ndarray

Notes

See Section III of [Cadambe2008] for details.

calc_Q(*k: int*) → numpy.ndarray
Calculates the interference covariance matrix at the k-th receiver.

The interference covariance matrix at the k-th receiver, k, is given by

$$k = \sum_{j=1, j \neq k}^{K} \frac{P_j}{Ns_j} {}_{jj}^{H} {}_{kj}^{H}$$

where P_j is the transmit power of transmitter j, and Ns_j is the number of streams for user j.

Parameters **k** (*int*) – Index of the desired receiver.

Returns **Qk** – The interference covariance matrix at receiver k.

Return type np.ndarray

Notes

This is impacted by the self.P attribute.

calc_Q_rev(*k: int*) → numpy.ndarray
Calculates the interference covariance matrix at the k-th receiver in the reverse network.

Parameters **k** (*int*) – Index of the desired receiver.

Returns **Qk_rev** – The interference covariance matrix at receiver k in the reverse network.

Return type np.ndarray

See also:

calc_Q()

calc_SINR() → numpy.ndarray
Calculates the SINR values (in linear scale) of all streams of all users with the current IA solution.

The noise variance used will be the value of the noise_var property, which, if not explicitly set, will use the noise_var property of the multiuserchannel object.

Returns **SINRs** – The SINR (in linear scale) of all streams of all users. This is a 1D numpy array of 1D numpy arrays (of floats).

Return type np.ndarray

calc_SINR_in_dB() → numpy.ndarray
Calculates the SINR values (in dB scale) of all streams of all users with the current IA solution.

The noise variance used will be the value of the noise_var property, which, if not explicitly set, will use the noise_var property of the multiuserchannel object.

Returns **SINRs** – The SINR (in dB scale) of all streams of all users. This is a 1D numpy array of 1D numpy arrays (of floats).

Return type np.ndarray

calc_SINR_old() → numpy.ndarray
Calculates the SINR values (in linear scale) of all streams of all users with the current IA solution.

The noise variance used will be the value of the noise_var property, which, if not explicitly set, will use the noise_var property of the multiuserchannel object.

This method is deprecated since it's not the correct way to calculate the SINR. Use the calc_SINR method instead.

Returns **SINRs** – The SINR (in linear scale) of all streams of all users. This is a 1D numpy array of 1D numpy arrays (of floats).

Return type np.ndarray

calc_remaining_interference_percentage(*k: int*, *Qk: Optional[numpy.ndarray] = None*) → float
Calculates the percentage of the interference in the desired signal space according to equation (30) in [Cadambe2008].

The percentage p_k of the interference in the desired signal space is given by

$$p_k = \frac{\sum_{j=1}^{Ns[k]} \lambda_j[k]}{Tr[k]}$$

where $\lambda_j[]$ denotes the j-th smallest eigenvalue of .

Parameters

- **k** (*int*) – The index of the desired user.
- **Qk** (*np.ndarray*) – The covariance matrix of the remaining interference at receiver k. If not provided, it will be automatically calculated. In that case, the *P* attribute will also be taken into account if it is set.

Returns The percentage of the interference in the desired signal space.

Return type float

Notes

Qk must be a symmetric matrix so that its eigenvalues are real and positive (any covariance matrix is a symmetric matrix).

calc_sum_capacity() → float
Calculates the sum capacity of the current solution.

The SINRs are estimated and applied to the Shannon capacity formula

Returns The sum capacity of the current solution.

Return type float

clear() → None
Clear the IA Solver object.

All member attributes that are updated during the solve method, such as the precoder and receive filters, will be cleared. The other attributes that correspond to "configuration" such as the channel object won't be changed.

Notes

You should overwrite this method in subclasses that pass parameters to the __init__ method, since here we call __init__ without arguments which is probably not what you want.

property full_F
Transmit precoder of all users.

Returns The precoders of all users (a 1D numpy array of 2D numpy arrays).

Return type np.ndarray

property full_W
Get method for the full_W property.

The full_W property returns the equivalent filter of the IA filter plus the post processing filter.

Returns the equivalent filter of the IA filter plus the post processing filter.

Return type np.ndarray

property full_W_H
Get method for the full_W_H property.

The full_W_H property returns the equivalent filter of the IA filter plus the post processing filter.

Returns The equivalent filter of the IA filter plus the post processing filter.

Return type np.ndarray

get_cost() → float
Get the current cost of the IA Solution.

This method should be implemented in subclasses and return a number greater than or equal to zero..

Returns **cost** – The Cost of the current IA solution (a real non-negative number).

Return type float

property noise_var
Get method for the noise_var property.

Returns The noise variance (a real non-negative number).

Return type float

randomizeF(*Ns: Union[int, List[int], Sequence[int]]*, *P: Optional[numpy.ndarray] = None*) → None
Generates a random precoder for each user.

Parameters

- **Ns** (*int | list[int] | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray, optional*) – Power of each user. If not provided, a value of 1 will be used for each user.

set_precoders(*F: Optional[Sequence[numpy.ndarray]] = None*, *full_F: Optional[Sequence[numpy.ndarray]] = None*, *P: Optional[numpy.ndarray] = None*) → None
Set the precoders of each user.

Either *F* or *full_F* (or both of them) must be provided.

If only *full_F* is provided then the value of *F* will be calculated from *full_F*.

In any case, the value of *self.Ns* will be updated according to the dimensions of the *F* or *full_F*.

Parameters

- **F** (*np.ndarray | list[np.ndarray], optional*) – A numpy array where each element is the (normalized) precoder (a 2D numpy array) of one user.
- **full_F** (*np.ndarray | list[np.ndarray], optional*) – A numpy array where each element is the precoder (a 2D numpy array) of one user.
- **P** (*np.ndarray, optional*) – The maximum transmit power. If not provided the current value of self.P will be kept as it is.

set_receive_filters(*W_H: Optional[Sequence[numpy.ndarray]] = None, W: Optional[Sequence[numpy.ndarray]] = None*) → None

Set the receive filters.

You only need to pass either *W_H* or *W*, but not both of them, since one is calculated from the other.

Parameters

- **W_H** (*np.ndarray | list[np.ndarray]*) – A numpy array where each element is the receive filter (a 2D numpy array) of one user. This is a 1D numpy array of 2D numpy arrays.
- **W** (*np.ndarray | list[np.ndarray]*) – A numpy array where each element is the receive filter (a 2D numpy array) of one user. This is a 1D numpy array of 2D numpy arrays.

abstract solve(*Ns: Union[int, numpy.ndarray], P: Optional[numpy.ndarray] = None*) → Any

Find the IA solution.

This method must be implemented in a subclass and should updates the 'F' and 'W' member variables.

Parameters

- **Ns** (*int | np.ndarray*) – Number of streams of each user.
- **P** (*np.ndarray*) – Power of each user. If not provided, a value of 1 will be used for each user.

Notes

This function should be implemented in the derived classes

Module contents

Package with Interference Alignment (IA) algorithms.

Note that all IA algorithms require the channel object and any change to the channel object must be performed before calling the *solve* method of the IA algorithm object. This includes generating the channel and setting the noise variance.

pyphysim.mimo package

Submodules

pyphysim.mimo.mimo module

Module implementing different MIMO schemes.

Each MIMO scheme is implemented as a class inheriting from *MimoBase* and implements at least the methods *encode*, *decode* and *getNumberOfLayers*.

class pyphysim.mimo.mimo.**Alamouti**(*channel: Optional[numpy.ndarray] = None*)
Bases: *pyphysim.mimo.mimo.MimoBase*

MIMO class for the Alamouti scheme.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

static _calc_precoder(*channel: numpy.ndarray*) → numpy.ndarray
Not defined.

There is no linear precoder for the Almost scheme and thus an exception is called if this method is ever called.

static _calc_receive_filter(*channel: numpy.ndarray*, *noise_var: Optional[float] = None*) → numpy.ndarray
Not defined.

There is no linear receive filter for the Alamouti scheme that can be directly applied to the received data. Thus, an exception is called if this method is ever called.

static _decode(*received_data: numpy.ndarray*, *channel: numpy.ndarray*) → numpy.ndarray
Perform the decoding of the received_data for the Alamouti scheme with the channel *channel*, but does not compensate for the power division among transmit antennas.

The idea is that the decode method will call _decode and perform the power compensation. This separation allows better code reuse.

Parameters

- **received_data** (*np.ndarray*) – Received data, which was encoded with the Alamouti scheme and corrupted by the channel *channel*.
- **channel** (*np.ndarray*) – MIMO channel matrix.

Returns **decoded_data** – The decoded data (without power compensating the power division performed during transmission).

Return type np.ndarray

See also:

decode()

static _encode(*transmit_data: numpy.ndarray*) → numpy.ndarray
Perform the Alamouti encoding, but without dividing the power among the transmit antennas.

The idea is that the encode method will call _encode and perform the power division. This separation allows better code reuse.

Parameters **transmit_data** (*np.ndarray*) – Data to be encoded by the Alamouti scheme.

Returns **encoded_data** – The encoded *transmit_data* (without dividing the power among transmit antennas).

Return type np.ndarray

See also:

encode()

calc_linear_SINRs(*noise_var: float*) → numpy.ndarray
Calculate the SINRs (in linear scale) of the multiple streams.

Parameters **noise_var** (*float*) – The noise variance.

Returns **sinrs** – The sinrs (in linear scale) of the multiple streams.

Return type np.ndarray

decode(*received_data: numpy.ndarray*) → numpy.ndarray
Perform the decoding of the received_data for the Alamouti scheme with the channel *channel*.

Parameters **received_data** (*np.ndarray*) – Received data, which was encoded with the Alamouti scheme and corrupted by the channel *channel*.

Returns **decoded_data** – The decoded data.

Return type np.ndarray

encode(*transmit_data: numpy.ndarray*) → numpy.ndarray
Perform the Alamouti encoding.

Parameters **transmit_data** (*np.ndarray*) – Data to be encoded by the Alamouti scheme.

Returns **encoded_data** – The encoded *transmit_data*.

Return type np.ndarray

getNumberOfLayers() → int
Get the number of layers of the Alamouti scheme.

The number of layers in the Alamouti scheme is always equal to one.

Returns **Nl** – Number of layers of the Alamouti scheme, which is always one.

Return type int

set_channel_matrix(*channel: numpy.ndarray*) → None
Set the channel matrix.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

Returns

Return type None

class pyphysim.mimo.mimo.**Blast**(*channel: Optional[numpy.ndarray] = None*)
Bases: *pyphysim.mimo.mimo.MimoBase*

MIMO class for the BLAST scheme.

The receive filter used will depend on the noise variance (see the *set_noise_var()* method). If the noise variance is positive the MMSE filter will be used, otherwise noise variance will be ignored and the Zero-Forcing filter will be used.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

static _calc_precoder(*channel: numpy.ndarray*) → numpy.ndarray
Calculate the linear precoder for the BLAST scheme.

The BLAST scheme simple send the data through the multiple streams without any particular precoding. Therefore, its linear precoder is equivalent to an identity matrix.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

Returns **W** – The precoder that can be applied to the input data.

Return type np.ndarray

static _calc_receive_filter(*channel: numpy.ndarray*, *noise_var: Optional[float] = None*) → numpy.ndarray
Calculate the receive filter for the MIMO scheme, if there is any.

Parameters

- **channel** (*np.ndarray*) – MIMO channel matrix.
- **noise_var** (*float*) – The noise variance. If a value is provided then MMSE filter will be used. If it is not provided (or None is passes) then Zero Force filter will be used.

Returns **G_H** – The receive_filter that can be applied to the input data.

Return type np.ndarray

decode(*received_data: numpy.ndarray*) → numpy.ndarray
Decode the received data array.

Parameters **received_data** (*np.ndarray*) – Received data, which was encoded with the Blast scheme and corrupted by the channel *channel*.

Returns **decoded_data** – The decoded data.

Return type np.ndarray

encode(*transmit_data: numpy.ndarray*) → numpy.ndarray
Encode the transmit data array to be transmitted using the BLAST scheme.

Parameters **transmit_data** (*np.ndarray*) – A numpy array with a number of elements which is a multiple of the number of transmit antennas.

Returns **encoded_data** – The encoded *transmit_data*.

Return type np.ndarray

Raises **ValueError** – If the number of elements in *transmit_data* is not multiple of the number of transmit antennas.

getNumberOfLayers() → int
Get the number of layers of the Blast scheme.

Returns **Nl** – Number of layers of the MIMO scheme.

Return type int

set_channel_matrix(*channel: numpy.ndarray*) → None
Set the channel matrix.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

set_noise_var(*noise_var: Optional[float]*) → None
Set the noise variance for the MMSE receive filter.

If noise_var is non-positive then the Zero-Force filter will be used instead.

Parameters **noise_var** (*float | None*) – Noise variance for the MMSE filter (if *noise_var* is positive). If *noise_var* is 0.0 or None then the Zero-Forcing filter will be used.

Returns

Return type None

class pyphysim.mimo.mimo.**GMDMimo**(*channel: Optional[numpy.ndarray] = None*)
Bases: *pyphysim.mimo.mimo.Blast*

MIMO class for the GMD based MIMO scheme.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

static _calc_precoder(*channel: numpy.ndarray*) → numpy.ndarray
Calculate the linear precoder for the GMD scheme.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix with dimension (1, Nt).

Returns **W** – The precoder that can be applied to the input data.

Return type np.ndarray

static _calc_receive_filter(*channel: numpy.ndarray*, *noise_var: Optional[float] = None*) → numpy.ndarray

Calculate the receive filter for the MRT scheme.

Parameters

- **channel** (*np.ndarray*) – MIMO channel matrix.
- **noise_var** (*float*) – The noise variance.

Returns **G_H** – The receive_filter that can be applied to the input data.

Return type np.ndarray

decode(*received_data: numpy.ndarray*) → numpy.ndarray

Perform the decoding of the received_data for the GMD MIMO.

Parameters **received_data** (*np.ndarray*) – Received data, which was encoded with the Alamouti scheme and corrupted by the channel *channel*.

Returns **decoded_data** – The decoded data.

Return type np.ndarray

encode(*transmit_data: numpy.ndarray*) → numpy.ndarray

Encode the transmit data array to be transmitted using the GMD MIMO scheme.

The GMD MIMO scheme is based on the Geometric Mean Decomposition (GMD) of the channel. The channel is decomposed into *H = Q R P^H*, where *R* is an upper triangular matrix with all diagonal elements being equal to the geometric mean of the singular values of the channel matrix *H*.

corresponds to using the 'U' and 'V' matrices from the SVD decomposition of the channel as the precoder and receive filter.

Parameters **transmit_data** (*np.ndarray*) – A numpy array with the data to be transmitted.

Returns **encoded_data** – The encoded *transmit_data*.

Return type np.ndarray

class pyphysim.mimo.mimo.**MRC**(*channel: Optional[numpy.ndarray] = None*)

Bases: *pyphysim.mimo.mimo.Blast*

MIMO class for the MRC scheme.

The receive filter used will depend on the noise variance (see the *set_noise_var()* method). If the noise variance is positive the MMSE filter will be used, otherwise noise variance will be ignored and the Zero-Forcing filter will be used.

The receive filter in the *Blast* class already does the maximum ratio combining. Therefore, this MRC class simply inherits from the Blast class and only exists for completion.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

set_channel_matrix(*channel: numpy.ndarray*) → None

Set the channel matrix.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix. The MRC MIMO scheme is defined for the scenario with multiple receive antennas and a single receive antenna. If channel is 1D assume that the number of transmit antennas is equal to 1.

class pyphysim.mimo.mimo.**MRT**(*channel: Optional[numpy.ndarray] = None*)

Bases: *pyphysim.mimo.mimo.MisoBase*

MIMO class for the MRT scheme.

The number of streams for the MRT scheme is always equal to one, but it still employs multiple transmit antennas.

If *channel* is not provided you need to call the *set_channel_matrix* method before calling the other methods.

Parameters **channel** (*np.ndarray*) – MISO channel vector. It must be a 1D numpy array, where the number of receive antennas is assumed to be equal to 1.

static _calc_precoder(*channel: numpy.ndarray*) → numpy.ndarray

Calculate the linear precoder for the MRT scheme.

The MRT scheme corresponds to multiplying the symbol from each transmit antenna with a complex number corresponding to the inverse of the phase of the channel so as to ensure that the signals add constructively at the receiver. This also means that the MRT scheme only be applied to scenarios with a single receive antenna.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix with dimension (1, Nt).

Returns **W** – The precoder that can be applied to the input data.

Return type np.ndarray

static _calc_receive_filter(*channel: numpy.ndarray*, *noise_var: Optional[float] = None*) → numpy.ndarray

Calculate the receive filter for the MRT scheme.

Parameters

- **channel** (*np.ndarray*) – MIMO channel matrix.
- **noise_var** (*float*) – The noise variance.

Returns **G_H** – The receive_filter that can be applied to the input data.

Return type np.ndarray

decode(*received_data: numpy.ndarray*) → numpy.ndarray

Decode the received data array.

Parameters **received_data** (*np.ndarray*) – Received data, which was encoded with the MRT scheme and corrupted by the channel *channel*.

Returns **decoded_data** – The decoded data.

Return type np.ndarray

encode(*transmit_data: numpy.ndarray*) → numpy.ndarray

Encode the transmit data array to be transmitted using the MRT scheme.

The MRT scheme corresponds to multiplying the symbol from each transmit antenna with a complex number corresponding to the inverse of the phase of the channel so as to ensure that the signals add constructively at the receiver. This also means that the MRT scheme only be applied to scenarios with a single receive antenna.

Parameters **transmit_data** (*np.ndarray*) – A numpy array with the data to be transmitted.

Returns **encoded_data** – The encoded *transmit_data*.

Return type np.ndarray

class pyphysim.mimo.mimo.**MimoBase**(*channel: Optional[numpy.ndarray] = None*)
Bases: object

Base Class for MIMO schemes.

All subclasses must implement at least the following methods:

- *getNumberOfLayers()*: Should return the number of layers of that specific MIMO scheme
- *encode()*: The encode method must perform everything executed at the transmitter for that specific MIMO scheme. This also include the power division among the transmit antennas.
- *decode()*: Analogous to the encode method, the decode method must perform everything performed at the receiver.

If possible, subclasses should implement the *_calc_precoder* and *_calc_receive_filter* static methods and use them in the implementation of *encode* and *decode*. This will allow using the *calc_linear_SINRs* and *calc_SINRs* methods to calculate the post processing SINRs. Note that calling the *_calcZeroForceFilter* and *_calcMMSE-Filter* methods in the implementation of the receive filter calculation can be useful.

If you can't implement the *_calc_precoder* and *_calc_receive_filter* static methods` (because there is no linear precoder or receive filter for the MIMO scheme in the subclass, for instance), then you should implement the *calc_linear_SINRs* method in the subclass instead.

Parameters **channel** (*np.ndarray | None*) – MIMO channel matrix. This should be a 1D or 2D numpy array. The allowed dimensions will depend on the particular MIMO scheme implemented in a subclass.

property Nr
Get the number of receive antennas

Returns The number of receive antennas.

Return type int

property Nt
Get the number of transmit antennas

Returns The number of transmit antennas.

Return type int

static _calcMMSEFilter(*channel: numpy.ndarray*, *noise_var: float*) → numpy.ndarray
Calculates the MMSE filter to cancel the inter-stream interference.

Parameters

- **channel** (*np.ndarray*) – MIMO channel matrix.
- **noise_var** (*float*) – Noise variance.

Returns **W** – The MMSE receive filter.

Return type np.ndarray

static _calcZeroForceFilter(*channel: numpy.ndarray*) → numpy.ndarray
Calculates the Zero-Force filter to cancel the inter-stream interference.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

Returns **W** – The Zero-Forcing receive filter.

Return type np.ndarray

Notes

The Zero-Force filter basically corresponds to the pseudo-inverse of the channel matrix.

static _calc_precoder(*channel: numpy.ndarray*) → numpy.ndarray
Calculate the linear precoder for the MIMO scheme, if there is any.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

Returns **W** – The precoder that can be applied to the input data.

Return type np.ndarray

static _calc_receive_filter(*channel: numpy.ndarray*, *noise_var: Optional[float] = None*) → numpy.ndarray
Calculate the receive filter for the MIMO scheme, if there is any.

Parameters

- **channel** (*np.ndarray*) – MIMO channel matrix.
- **noise_var** (*float*) – The noise variance.

Returns **G_H** – The receive_filter that can be applied to the input data.

Return type np.ndarray

calc_SINRs(*noise_var: float*) → numpy.ndarray
Calculate the SINRs (in dB) of the multiple streams.

Parameters **noise_var** (*float*) – The noise variance.

Returns **SINRs** – The SINRs (in dB) of the multiple streams.

Return type np.ndarray

calc_linear_SINRs(*noise_var: float*) → numpy.ndarray
Calculate the SINRs (in linear scale) of the multiple streams.

Parameters **noise_var** (*float*) – The noise variance.

Returns **sinrs** – The sinrs (in linear scale) of the multiple streams.

Return type np.ndarray

abstract decode(*received_data: numpy.ndarray*) → numpy.ndarray
Method to decode the transmit data array to be transmitted using some MIMO scheme. This method must be implemented in a subclass.

Parameters **received_data** (*np.ndarray*) – The received data.

Returns **decoded_data** – The decoded data.

Return type np.ndarray

abstract encode(*transmit_data: numpy.ndarray*) → numpy.ndarray
Method to encode the transmit data array to be transmitted using some MIMO scheme. This method must be implemented in a subclass.

Parameters **transmit_data** (*np.ndarray*) – The data to be transmitted.

Returns **encoded_data** – The encoded *transmit_data*.

Return type np.ndarray

abstract getNumberOfLayers() → int
Get the number of layers of the MIMO scheme.

Notes

This method must be implemented in each subclass of *MimoBase*.

Returns The number of layers.

Return type int

set_channel_matrix(*channel: numpy.ndarray*) → None
Set the channel matrix.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix. This should be a 1D or 2D numpy array. The allowed dimensions will depend on the particular MIMO scheme implemented in a subclass.

class pyphysim.mimo.mimo.**MisoBase**(*channel: Optional[numpy.ndarray] = None*)
Bases: *pyphysim.mimo.mimo.MimoBase*

Base Class for MISO schemes.

All subclasses must implement at least the following methods:

- *MimoBase.encode()*: The encode method must perform everything executed at the transmitter for that specific MIMO scheme. This also include the power division among the transmit antennas.
- *MimoBase.decode()*: Analogous to the encode method, the decode method must perform everything performed at the receiver.

Other optional methods that might be useful implementing in subclasses are the *_calc_precoder* and *_calc_receive_filter* methods.

Parameters **channel** (*np.ndarray*) – MISO channel matrix/vector. MISO schemes are defined for scenarios with multiple transmit antennas and a single receive antenna. If *channel* is 2D, then the first dimension size must be equal to 1.

getNumberOfLayers() → int
Get the number of layers of the MISO scheme.

Because a MISO scheme only has one receive antenna then then number of layers is always equal to 1.

Returns The number of layers.

Return type int

set_channel_matrix(*channel: numpy.ndarray*) → None
Set the channel matrix.

Parameters **channel** (*np.ndarray*) – MISO channel vector. A MISO scheme is defined for the scenario with multiple transmit antennas and a single receive antenna. If channel is 2D then the first dimension size must be equal to 1.

Returns

Return type None

class pyphysim.mimo.mimo.**SVDMimo**(*channel: Optional[numpy.ndarray] = None*)
Bases: *pyphysim.mimo.mimo.Blast*

MIMO class for the SVD MIMO scheme.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix.

static _calc_precoder(*channel: numpy.ndarray*) → numpy.ndarray
Calculate the linear precoder for the SVD MIMO scheme.

The SVD MIMO scheme employs as precoder the right singular matrix from SVD (Singular Value Decomposition) of the channel.

Parameters **channel** (*np.ndarray*) – MIMO channel matrix with dimension (1, Nt).

Returns **W** – The precoder that can be applied to the input data.

Return type np.ndarray

static _calc_receive_filter(*channel: numpy.ndarray*, *noise_var: Optional[float] = None*) → numpy.ndarray

Calculate the receive filter for the SVD MIMO scheme.

Parameters

- **channel** (*np.ndarray*) – MIMO channel matrix.
- **noise_var** (*float*) – The noise variance.

Returns **G_H** – The receive_filter that can be applied to the input data.

Return type np.ndarray

decode(*received_data: numpy.ndarray*) → numpy.ndarray

Perform the decoding of the received_data for the SVD MIMO scheme with the channel *channel*.

Parameters **received_data** (*np.ndarray*) – Received data, which was encoded with the Alamouti scheme and corrupted by the channel *channel*.

Returns **decoded_data** – The decoded data.

Return type np.ndarray

encode(*transmit_data: numpy.ndarray*) → numpy.ndarray

Encode the transmit data array to be transmitted using the SVD MIMO scheme.

The SVD MIMO scheme corresponds to using the 'U' and 'V' matrices from the SVD decomposition of the channel as the precoder and receive filter.

Parameters **transmit_data** (*np.ndarray*) – A numpy array with the data to be transmitted.

Returns **encoded_data** – The encoded *transmit_data*.

Return type np.ndarray

Module contents

pyphysim.modulators package

Submodules

pyphysim.modulators.fundamental module

Module with class for some fundamental modulators, such as PSK and M-QAM.

All fundamental modulators inherit from the *Modulator* class and should call the self.setConstellation method in their __init__ method, as well as implement the calcTheoreticalSER and calcTheoreticalBER methods.

class pyphysim.modulators.fundamental.**BPSK**

Bases: *pyphysim.modulators.fundamental.Modulator*

BPSK Class

calcTheoreticalBER(*SNR: NumberOrArray*) → NumberOrArray
Calculates the theoretical (approximation) bit error rate for the BPSK scheme.

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **BER** – The theoretical bit error rate.

Return type float | np.ndarray

calcTheoreticalSER(*SNR: NumberOrArray*) → NumberOrArray
Calculates the theoretical (approximation) symbol error rate for the BPSK scheme.

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **SER** – The theoretical symbol error rate.

Return type float | np.ndarray

demodulate(*receivedData: numpy.ndarray*) → numpy.ndarray
Demodulate the data.

Parameters **receivedData** (*np.ndarray*) – Data to be demodulated.

Returns **demodulated_data** – The demodulated data.

Return type np.ndarray

modulate(*inputData: numpy.ndarray*) → numpy.ndarray
Modulate the input data (decimal data).

Parameters **inputData** (*np.ndarray*) – Data to be modulated.

Returns **modulated_data** – The modulated data

Return type np.ndarray

Raises **ValueError** – If inputData has any invalid value such as values greater than self._M - 1. Note that inputData should not have negative values but no check is done for this.

property name
Get the name property.

Returns The name of the modulator.

Return type str

class pyphysim.modulators.fundamental.**Modulator**
Bases: object

Base class for digital modulators.

The derived classes need to at least call setConstellation to set the constellation in their constructors as well as implement calcTheoreticalSER and calcTheoreticalBER.

Examples

```
>>> np.set_printoptions(linewidth=70)
>>> constellation = np.array([1 + 1j, - 1 + 1j, - 1 - 1j, 1 - 1j])
>>> m=Modulator()
>>> m.setConstellation(constellation)
>>> m.symbols
array([ 1.+1.j, -1.+1.j, -1.-1.j,  1.-1.j])
>>> m.M
4
```

(continues on next page)

(continued from previous page)

```
>>> m.K
2.0
>>> m
4-Modulator object
>>> m.modulate(np.array([0, 0, 3, 3, 1, 3, 3, 3, 2, 2]))
array([ 1.+1.j,  1.+1.j,  1.-1.j,  1.-1.j, -1.+1.j,  1.-1.j,  1.-1.j,
        1.-1.j, -1.-1.j, -1.-1.j])
```

```
>>> m.demodulate(np.array([ 1. + 1.j, 1. + 1.j, 1. - 1.j, 1. - 1.j,
↪                   - 1. + 1.j, 1. - 1.j, 1. - 1.j, 1. - 1.j,
↪             - 1. - 1.j, - 1. - 1.j]))
array([0, 0, 3, 3, 1, 3, 3, 3, 2, 2])
```

property K

Get method for the K property.

The *K* property corresponds to the number of bits represented by each symbol in the constellation. It is equal to log2(M), where *M* is the constellation size.

See also:

M

property M

Get method for the M property.

The *M* property corresponds to the number of symbols in the constellation.

See also:

K

calcTheoreticalBER(*SNR: NumberOrArray*) → NumberOrArray

Calculates the theoretical bit error rate.

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **BER** – The theoretical bit error rate.

Return type float | np.ndarray

See also:

calcTheoreticalSER(), *calcTheoreticalPER()*

Notes

This function should be implemented in the derived classes

calcTheoreticalPER(*SNR: NumberOrArray*, *packet_length: int*) → NumberOrArray

Calculates the theoretical package error rate.

A package is a group of bits, where if a single bit is in error then the whole package is considered to be in error.

The package error rate (PER) is a direct mapping of the bit error rate (BER), such that

$$PER = 1 - (1 - BER)^L$$

where L is the package_length.

Parameters

- **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).
- **packet_length** (*int*) – The package length. That is, the number of bits in each package.

Returns **PER** – The theoretical package error rate.

Return type float | np.ndarray

See also:

calcTheoreticalBER(), *calcTheoreticalSER()*, *calcTheoreticalSpectralEfficiency()*

calcTheoreticalSER(*SNR: NumberOrArray*) → NumberOrArray

Calculates the theoretical symbol error rate.

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **SER** – The theoretical symbol error rate.

Return type float | np.ndarray

See also:

calcTheoreticalBER(), *calcTheoreticalPER()*

Notes

This function should be implemented in the derived classes

calcTheoreticalSpectralEfficiency(*SNR: NumberOrArray*, *packet_length: Optional[int] = None*) → NumberOrArray

Calculates the theoretical spectral efficiency.

If there was no error in the transmission, the spectral efficiency would be equal to the *K* property, that is, equal to the number of bits represented by each symbol in the constellation. However, due to bit errors the effective spectral efficiency will be lower.

The calcTheoreticalSpectralEfficiency method calculates the effective spectral efficiency from the *K* property and the package error rate (PER) for the given SNR and packet_length 'L', such that

$$se = K * (1 - PER)$$

Parameters

- **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).
- **packet_length** (*int, optional*) – The package length. That is, the number of bits in each package.

Returns **se** – The theoretical spectral efficiency.

Return type float | np.ndarray

See also:

calcTheoreticalBER(), *calcTheoreticalPER()*, *K()*

demodulate(*receivedData: numpy.ndarray*) → numpy.ndarray

Demodulate the data.

Parameters **receivedData** (*np.ndarray*) – Data to be demodulated.

Returns **demodulated_data** – The demodulated data.

Return type np.ndarray

modulate(*inputData: Union[int, numpy.ndarray]*) → numpy.ndarray
Modulate the input data (decimal data).

Parameters **inputData** (*np.ndarray | int*) – Data to be modulated.

Returns **modulated_data** – The modulated data

Return type np.ndarray

Raises **ValueError** – If inputData has any invalid value such as values greater than self._M - 1. Note that inputData should not have negative values but no check is done for this.

property name
Get method for the 'name' property.

Returns The name of the modulator.

Return type str

plotConstellation() → None
Plot the constellation (in a scatter plot).

setConstellation(*symbols: numpy.ndarray*) → None
Set the constellation of the modulator.

This function should be called in the constructor of the derived classes.

Parameters **symbols** (*np.ndarray*) – A an numpy array with the symbol table.

class pyphysim.modulators.fundamental.**PSK**(*M: int*, *phaseOffset: float = 0*)
Bases: *pyphysim.modulators.fundamental.Modulator*

PSK Class

static _createConstellation(*M: int*, *phaseOffset: float*) → numpy.ndarray
Generates the Constellation for the PSK modulation scheme.

Parameters

- **M** (*int*) – The modulation cardinality
- **phaseOffset** (*float*) – A phase offset (in radians) to be applied to the PSK constellation.

Returns **symbols** – The PSK constellation with the desired cardinality and phase offset.

Return type np.ndarray

calcTheoreticalBER(*SNR: NumberOrArray*) → NumberOrArray
Calculates the theoretical (approximation) bit error rate for the M-PSK scheme using Gray coding.

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **BER** – The theoretical bit error rate.

Return type float | np.ndarray

calcTheoreticalSER(*SNR: NumberOrArray*) → NumberOrArray
Calculates the theoretical (approximation for high M and high SNR) symbol error rate for the M-PSK scheme.

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **SER** – The theoretical symbol error rate.

Return type float | np.ndarray

setPhaseOffset (*phaseOffset: float*) → None
Set a new phase offset for the constellation

Parameters **phaseOffset** (*float*) – A phase offset (in radians) to be applied to the PSK constellation.

class pyphysim.modulators.fundamental.**QAM**(*M: int*)
Bases: *pyphysim.modulators.fundamental.Modulator*

QAM Class

_calcTheoreticalSingleCarrierErrorRate (*SNR: NumberOrArray*) → NumberOrArray
Calculates the theoretical (approximation) error rate of a single carrier in the QAM system (QAM has two carriers).

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **Psc** – The theoretical single carrier error rate.

Return type float | np.ndarray

Notes

This method is used in the *calcTheoreticalSER()* implementation.

See also:

calcTheoreticalSER()

static _calculateGrayMappingIndexQAM (*L: int*) → numpy.ndarray
Calculates the indexes that should be applied to the constellation created by _createConstellation in order to correspond to Gray mapping.

Notice that the square M-QAM constellation is a matrix of dimension L x L, where L is the square root of M. Since the constellation was generated without taking into account the Gray mapping, then we need to reorder the generated constellation and this function calculates the indexes that can be applied to the original constellation in order to do exactly that.

As an example, for the 16-QAM modulation the indexes can be organized (row order) in the matrix below

/	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

This is equivalent to concatenate a Gray mapping for the row with a Gray mapping for the column, and the corresponding indexes are [0, 1, 3, 2, 4, 5, 7, 6, 12, 13, 15, 14, 8, 9, 11, 10]

Parameters **L** (*int*) – Square root of the modulation cardinality (must be an integer).

Returns **indexes** – indexes that should be applied to the constellation created by _createConstellation in order to correspond to Gray mapping

Return type np.ndarray

static _createConstellation (*M: int*) → numpy.ndarray
Generates the Constellation for the (SQUARE) M-QAM modulation scheme.

Parameters **M** (*int*) – The modulation cardinality

Returns **symbols** – The QAM constellation with the desired cardinality.

Return type np.ndarray

calcTheoreticalBER(*SNR: NumberOrArray*) → NumberOrArray
Calculates the theoretical (approximation) bit error rate for the QAM scheme.

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **BER** – The theoretical bit error rate.

Return type float | np.ndarray

calcTheoreticalSER(*SNR: NumberOrArray*) → NumberOrArray
Calculates the theoretical (approximation) symbol error rate for the QAM scheme.

Parameters **SNR** (*float | np.ndarray*) – Signal-to-noise-value (in dB).

Returns **SER** – The theoretical symbol error rate.

Return type float | np.ndarray

class pyphysim.modulators.fundamental.**QPSK**
Bases: *pyphysim.modulators.fundamental.PSK*

QPSK Class

pyphysim.modulators.ofdm module

Module implementing OFDM modulation and demodulation.

class pyphysim.modulators.ofdm.**OFDM**(*fft_size: int*, *cp_size: int*, *num_used_subcarriers: Optional[int] = None*)
Bases: object

OFDM class.

_add_CP(*input_data: numpy.ndarray*) → numpy.ndarray
Add the Cyclic prefix to the input data.

Parameters **input_data** (*np.ndarray*) – OFDM modulated data (after the IFFT). This must be a 2D numpy array with shape (Number of OFDM symbols, IFFT size).

Returns **output** – The *input_data* with the cyclic prefix added. The shape of the output is (Number of OFDM symbols, IFFT size + CP Size).

Return type np.ndarray

_calc_zeropad(*input_data_size: int*) → Tuple[int, int]
Calculates the number of zeros that must be added to the input data to make it a multiple of the OFDM size.

The number of zeros that must be added to the input data is returned along with the number of OFDM symbols that will be generated.

Parameters **input_data_size** (*int*) – Size the the data that will be modulated by the OFDM object.

Returns **(zeropad, num_ofdm_symbols)** – A tuple with zeropad and num_ofdm_symbols. Zeropad is the number of zeros added to the input data to make the total number of elements a multiple of the number of used subcarriers. Num_ofdm_symbols is the number of OFDM symbols required to transmit *input_data_size* symbols.

Return type tuple[int,int]

_calculate_power_scale() → float
Calculate the power scale that needs to be applied in the modulator and removed in the demodulate methods.

The power is applied in the modulator method so that the total power of the OFDM samples is similar to the total power of the symbols modulated by OFDM.

Note that this total power is shared among useful samples and the cyclic prefix in one OFDM symbol. Therefore, the larger the cyclic prefix size the lower is this power scale to account energy loss due to sending the cyclic prefix.

Returns **power_scale** – The calculated power scale. You should take the square root of this before multiplying by the samples.

Return type float

_get_subcarrier_numbers() → numpy.ndarray
Get the indexes of all subcarriers, including the negative, the DC and the positive subcarriers.

Note that these indexes are not suitable for indexing in python. They are the actual indexes of the subcarriers in an OFDM symbol. For instance, an OFDM symbol with 16 subcarriers will have indexes from -8 to 7. However, due to the way the fft is implemented in numpy the indexes here are actually from 0 to 7 followed by -8 to -1.

Returns Numbers of all subcarriers, including the negative, the DC and the positive subcarriers

Return type np.ndarray

Examples

>> ofdm_obj = OFDM(16, 4, 16) >> ofdm_obj._get_subcarrier_numbers() array([0, 1, 2, 3, 4, 5, 6, 7, -8, -7, -6, -5, -4, -3, -2, -1])

_get_used_subcarrier_numbers() → numpy.ndarray
Get the subcarrier indexes of the actually used subcarriers.

Note that these indexes are not suitable for indexing in python. They are the actual indexes of the subcarriers in an OFDM symbol. See the documentation of the _get_subcarrier_numbers function.

Returns Number of the actually used subcarriers.

Return type np.ndarray

Examples

>> ofdm_obj = OFDM(16, 4, 10) >> ofdm_obj._get_used_subcarrier_numbers() array([1, 2, 3, 4, 5, -5, -4, -3, -2, -1]) >> ofdm_obj = OFDM(16, 4, 14) >> ofdm_obj._get_used_subcarrier_numbers() array([1, 2, 3, 4, 5, 6, 7, -7, -6, -5, -4, -3, -2, -1])

_prepare_decoded_signal(*decoded_signal: numpy.ndarray*) → numpy.ndarray
Prepare the decoded signal that was processed by the FFT in the demodulate function.

This is equivalent of reversing the indexing that was done by the _prepare_input_signal method.

Parameters **decoded_signal** (*np.ndarray*) – Signal that was decoded by the FFT in the OFDM demodulate method.

Returns **demodulated_samples** – Demodulated samples of the symbols that were modulated by the OFDM object (for instance the PSK or M-QAM symbols passed to OFDM).

Return type np.ndarray

Notes

This method should be called AFTER the Cyclic Prefix was removed and the FFT was performed.

Also, because the number of zeropad was not saved, then _prepare_decoded_signal has no way to remove them.

See also:

_prepare_input_signal()

_prepare_input_signal (*input_signal: numpy.ndarray*) → numpy.ndarray
Prepare the input signal to be passed to the IFFT in the modulate function.

The input signal must be prepared before it is passed to the IFFT in the OFDM modulate function.

- First, zeros must be added so that the input signal size is multiple of the number of used subcarriers.
- After that the input signal must be allocated to subcarriers in the center of the spectrum (except for the DC component). That is, zeros will be allocated to the lower and higher subcarriers such that only num_used_subcarriers are used from fft_size subcarriers.

This preparation is performed by the _prepare_input_signal function.

Parameters **input_signal** (*np.ndarray*) – Input signal that must be modulated by the OFDM modulate function.

Returns **input_ifft** – Signal suitable to be passed to the IFFT function to actually perform the OFDM modulation.

Return type np.ndarray

See also:

_prepare_decoded_signal()

_remove_CP (*received_data: numpy.ndarray*) → numpy.ndarray
Remove the Cyclic prefix of the received data.

Parameters **received_data** (*np.ndarray*) – Data that must be demodulated by the OFDM object.

Returns **output** – Received data without the Cyclic prefix.

Return type np.ndarray

Notes

The _remove_CP method will also change the shape so that it is suitable to be passed to the FFT function.

demodulate (*received_signal: numpy.ndarray*) → numpy.ndarray
Perform the OFDM demodulation of the received_signal.

Parameters **received_signal** (*np.ndarray*) – An array with the samples of the received OFDM symbols.

Returns **demodulated_data** – Demodulated symbols.

Return type np.ndarray

get_used_subcarrier_indexes () → numpy.ndarray
Get the subcarrier indexes of the subcarriers actually used in a way suitable for python indexing (going from 0 to fft_size-1).

Returns indexes – Subcarrier indexes of the subcarriers actually used in a way suitable for python indexing.

Return type np.ndarray

Notes

This is the function actually used in the modulate function.

Examples

Consider the example below where we have 16 subcarriers and only 10 subcarriers are used. The lower and higher subcarrier as well as the DC subcarrier will not be used. The index of the used subcarriers should go then from 11 to 15 (5 subcarriers), skip subcarrier 0, and then go from 1 to 5 (the other 5 subcarriers).

```
>>> ofdm_obj = OFDM(16, 4, 10)
>>> ofdm_obj.get_used_subcarrier_indexes()
array([11, 12, 13, 14, 15,  1,  2,  3,  4,  5])
>>> ofdm_obj = OFDM(16,4,14)
>>> ofdm_obj.get_used_subcarrier_indexes()
array([ 9, 10, 11, 12, 13, 14, 15,  1,  2,  3,  4,  5,  6,  7])
```

modulate(*input_signal: numpy.ndarray*) → numpy.ndarray
Perform the OFDM modulation of the input_signal.

Parameters **input_signal** (*np.ndarray*) – Input signal that must be modulated by the OFDM modulate function.

Returns **output** – An array with the samples of the modulated OFDM symbols.

Return type np.ndarray

set_parameters(*fft_size: int*, *cp_size: int*, *num_used_subcarriers: Optional[int] = None*) → None
Set the OFDM parameters.

Parameters

- **fft_size** (*int*) – Size of the FFT and IFFT used by the OFDM class.
- **cp_size** (*int*) – Size of the cyclic prefix (in samples).
- **num_used_subcarriers** (*int, optional*) – Number of used subcarriers. Must be greater than or equal to 2 and lower than or equal to fft_size. If not provided, fft_size will be used

Raises **ValueError** – If the any of the parameters are invalid.

class pyphysim.modulators.ofdm.**OfdmOneTapEqualizer**(*ofdm_obj: pyphysim.modulators.ofdm.OFDM*)

Bases: object

The OfdmOneTapEqualizer class performs the one-tap equalization often required in OFDM transmissions to compensate the effect of the channel at each subcarrier.

Parameters **ofdm_obj** (OFDM) – The OFDM object used to modulate/demodulate the data.

_equalize_data(*data_reshaped: numpy.ndarray*, *mean_freq_response: numpy.ndarray*) → numpy.ndarray
Perform the one-tap equalization and return *data* after the channel compensation.

Parameters

- **data_reshaped** (*np.ndarray*) – The data to be equalized. If must be a 2D numpy array, where different rows correspond to different OFDM symbols and the different columns correspond to the USED subcarriers. Dimension: *num OFDM symbols x num Used subcarriers*
- **mean_freq_response** (*np.ndarray*) – The frequency response for each OFDM symbol. Dimension: *num OFDM symbols x FFT size*

Returns The received *data* after the one-tap equalization to compensate the channel effect. Dimension: *num OFDM symbols x num Used subcarriers*

Return type np.ndarray

equalize_data(*data: numpy.ndarray*, *impulse_response:* pyphysim.channels.fading.TdlImpulseResponse) → numpy.ndarray

Perform the one-tap equalization and return *data* after the channel compensation.

Parameters

- **data** (*np.ndarray*) – The data to be equalized.
- **impulse_response** (fading.TdlImpulseResponse) – The impulse response of the channel.

Returns The received *data* after the one-tap equalization to compensate the channel effect.

Return type np.ndarray

Module contents

Package containing modulators.

pyphysim.pointprocess package

Submodules

pyphysim.pointprocess.pointprocess module

pyphysim.pointprocess.pointprocess.**generate_random_points_in_circle**(*num_points: int*, *max_radius: float*, *min_radius: float = 0.0*) → numpy.ndarray

Generate *numPoints* points uniformly inside a circle of radius *max_radius* and outside a circle with radius *min_radius*

The circle center is at the origin.

Parameters

- **num_points** – The desired number of points
- **max_radius** – The circle radius
- **min_radius** – The minimum radius

Returns The random points inside the circle.

Return type np.ndarray

pyphysim.pointprocess.pointprocess.**generate_random_points_in_rectangle**(*num_points: int*, *width: float*, *height: float*)

Generate *num_points* points uniformly inside a rectangle.

The rectangle center is at the origin.

Parameters

- **num_points** – The desired number of points
- **width** – The rectangle width
- **height** – The rectangle height

Returns The random points inside the circle.

Return type np.ndarray

Module contents

pyphysim.progressbar package

Submodules

pyphysim.progressbar.progressbar module

Implement classes to represent the progress of a task.

Use the ProgressbarText class for tasks that do not use multiprocessing, and the ProgressbarMultiProcessServer class for tasks using multiprocessing.

Basically, the task code must call the "progress" function to update the progress bar and pass a number equivalent to the increment in the progress since the last call. The progressbar must know the maximum value equivalent to all the progress, which is passed during object creator for ProgressbarText class.

The ProgressbarMultiProcessServer is similar to ProgressbarText class, accounts for the progress of multiple processes. For each process you need to call the register_client_and_get_proxy_progressbar to get a proxy progressbar, where the maximum value equivalent to all the progress that will be performed by that process is passed in this proxy creation. Each process then calls the progress method of the proxy progressbar.

Note that there is also a DummyProgressbar whose progress function does nothing. This is useful when you want to give the user a choice to show or not the progressbar such that the task code can always call the progress method and you only change the progressbar object.

class pyphysim.progressbar.progressbar.**DummyProgressbar**(**args: Any*, ***kwargs: Any*)

Bases: object

Dummy progress bar that don't really do anything.

The idea is that it can be used in place of the *ProgressbarText* class, but without actually doing anything.

See also:

ProgressbarText

progress(*count: int*) → None
This *progress* method has the same signature from the one in the *ProgressbarText* class.

Nothing happens when this method is called.

Parameters **count** (*int*) – Ignored

class pyphysim.progressbar.progressbar.**ProgressBarBase**(*finalcount: int*)
Bases: object

Base class for all ProgressBar classes.

Parameters **finalcount** (*int*) – The total amount that corresponds to 100%. Each time the progress method is called with a number that number is added with the current amount in the progressbar. When the amount becomes equal to *finalcount* the bar will be 100% complete.

Notes

Derived classes should implement *_update_iteration()* and *_display_current_progress()*. Optionally derived class might also implement *_perform_initialization()* and *_perform_finalizations()*

_count_to_percent(*count: int*) → float
Convert a given count into the equivalent percentage.

Parameters **count** (*int*) – The current count to be represented in the progressbar. The progressbar represents this count as a percent value of self.finalcount

Returns **percentage** – The percentage that *count* is of self.finalcount (between 0 and 100)

Return type float

_display_current_progress() → None
Refresh the progress representation.

This method should be defined in a subclass.

_perform_finalizations() → None
Perform any finalization (cleanings) after the progressbar stops.

This method should be implemented in sub-classes if any finalization code should be run.

_perform_initialization() → None
Perform any initializations for the progressbar.

This method should be implemented in sub-classes if any initialization code should be run.

_update_iteration(*count: int*) → None
Update the progressbar according with the new *count*.

Parameters **count** (*int*) – The current count to be represented in the progressbar. The progressbar represents this count as a percent value of self.finalcount

property elapsed_time
Get method for the elapsed_time property.

Returns The elapsed time.

Return type str

final progress(*count: int*) → None
Updates the current progress.

Calling this function will update the the current progress.

Parameters **count** (*int*) – The current count to be represented in the progressbar. The progressbar represents this count as a percent value of self.finalcount

Notes

How the progressbar is actually represented depends on the subclass. In the subclasses implement the *_update_iteration* method to update the current representation of the progressbar and the *_update_progress_display* to actually display the current progress.

final start() → None
Start the progressbar.

This method should be called just before the progressbar is used for the first time. Among possible other things, it will store the current time so that the elapsed time can be tracked.

If is automatically called in the *progress* method, if not called before.

final stop() → None
Stop the progressbar.

This method is automatically called in the *progress* method when the progress reaches 100%. If manually called, any subsequent call to the *progress* method will be ignored.

class pyphysim.progressbar.progressbar.**ProgressBarIPython**(*finalcount: int*, *side_message: Optional[str] = None*)
Bases: *pyphysim.progressbar.progressbar.ProgressBarBase*

Progressbar for IPython notebooks.

The progressbar will be rendered using IPython widgets.

_display_current_progress() → None
Refresh the progress representation.

_perform_initialization() → None
Perform any initializations for the progressbar.

This method should be implemented in sub-classes if any initialization code should be run.

_update_iteration(*count: int*) → None
Update the self.prog_bar member variable according with the new *count*.

Parameters **count** (*int*) – The current count to be represented in the progressbar. The progressbar represents this count as a percent value of self.finalcount

class pyphysim.progressbar.progressbar.**ProgressbarDistributedClientBase**(*client_id: int*, *finalcount: int*)
Bases: *pyphysim.progressbar.progressbar.ProgressBarBase*

Proxy progressbar that behaves like a ProgressbarText object, but is actually updating a shared (with other clients) progressbar.

The basic idea is that this proxy progressbar has the "progress" method similar to the standard ProgressbarText class. However, when this method is called it will update a value that will be read by a "server progressbar" object which is responsible to actually show the current progress.

Parameters **client_id** (*ClientID*) – The client ID.

_display_current_progress() → None
Refresh the progress representation.

This method should be defined in a subclass.

_update_iteration(*count: int*) → None
Update the progressbar according with the new *count*.

Parameters **count** (*int*) – The current count to be represented in the progressbar. The progressbar represents this count as a percent value of self.finalcount

class pyphysim.progressbar.progressbar.**ProgressbarDistributedServerBase**(*progresschar: str = '*', message: str = '', sleep_time: float = 1.0, filename: Optional[str] = None, style='text2'*)

Bases: object

Base class for progressbars for distributed computations.

In order to track the progress of distributed computations two classes are required, one that acts as a central point and is responsible to actually show the progress (the server), and other class that acts as a proxy (the client) and is responsible to sending the current progress to the server. There will be one object of the "server class" and one or more objects of the "client class", each one tracking the progress of one of the distributed computations.

This class is a base class for the "server part", while the *ProgressbarDistributedClientBase* class is a base class for the "client part".

For a full implementation, see the *ProgressbarMultiProcessServer* and *ProgressbarMultiProcessClient* classes.

Parameters

- **progresschar** (*str*) – Character used in the progressbar.
- **message** (*str*) – Message writen in the progressbar.
- **sleep_time** (*float*) – Time between progressbar updates (in seconds).

- **filename** (*str*) – If filename is None (default) then progress will be output to sys.stdout. If it is not None then the progress will be output to a file with name *filename*. This is usually useful for debugging and testing purposes.
- **style** (*str*) – The progressbar style. It controls which progressbar is used to display progress. It can be either 'text1', 'text2', 'text3', or 'ipython'

_ProgressbarDistributedServerBase__cretae_inner_progressbar(*output*)

_register_client(*total_count: int*) → int
Register a new "client" for the progressbar and return its *client_id*.

These returned values must be passed to the corresponding proxy progressbar.

Parameters **total_count** (*int*) – Total count that will be equivalent to 100% progress for the function.

Returns **client_id** – The client_id. The function whose process is tracked by the ProgressbarMultiProcessServer must update the element *client_id* with the current count.

Return type ClientID

_update_client_data_list() → None
This method process the communication between the client and the server.

It should gather the information sent by the clients (proxy progressbars) and update the member variable self._client_data_list accordingly, which will then be automatically represented in the progressbar.

_update_progress(*filename: Optional[str] = None*, *start_delay: float = 0.0*) → None
Collects the progress from each registered proxy progressbar and updates the actual visible progressbar.

Parameters

- **filename** (*str*) – Name of a file where the data will be written to. If this is None then all progress will be printed in the standard output (default)
- **start_delay** (*float, optional*) – Delay in seconds before starting the progressbar. During this time it is still possible to register new clients and the progressbar will only be shown after this delay..

register_client_and_get_proxy_progressbar(*total_count: int*) → *pyphysim.progressbar.progressbar.ProgressbarDistributedClientBase*
Register a new "client" for the progressbar and returns a new proxy progressbar that the client can use to update its progress by calling the *progress* method of this proxy progressbar.

Parameters **total_count** (*int*) – Total count that will be equivalent to 100% for function.

Returns **obj** – The proxy progressbar.

Return type Object of a class derived from ProgressbarDistributedClientBase

start_updater(*start_delay: float = 0.0*) → None
Start the process that updates the progressbar.

Parameters **start_delay** (*float*) – Delay in seconds before starting the progressbar. During this time it is still possible to register new clients and the progressbar will only be shown after this delay..

Notes

If this method is called multiple times then the *stop_updater* method must be called the same number of times for the updater process to actually stop.

stop_updater(*timeout: Optional[float] = None*) → None
Stop the process updating the progressbar.

You should always call this function in your main process (the same that created the progressbar) after joining all the processes that update the progressbar. This guarantees that the progressbar updated any pending change and exited clearly.

Parameters **timeout** (*float*) – The timeout to join for the process to stop.

Notes

If the *start_updater* was called multiple times the process will only be stopped when *stop_updater* is called the same number of times.

property total_final_count
Get method for the total_final_count property.

Returns The final count.

Return type int

class pyphysim.progressbar.progressbar.**ProgressbarMultiProcessClient**(*client_id: int*, *client_data_list: List[Any]*, *final_count: int*)
Bases: *pyphysim.progressbar.progressbar.ProgressbarDistributedClientBase*

Proxy progressbar that behaves like a ProgressbarText object, but is actually updating a ProgressbarMultiProcessServer progressbar.

The basic idea is that this proxy progressbar has the "progress" method similar to the standard ProgressbarText class. However, when this method is called it will update a value that will be read by a ProgressbarMultiProcessServer object instead.

Parameters

- **client_id** (*ClientID*) – The client ID
- **client_data_list** (*list*) – The client data list

_update_iteration(*count: int*) → None
Updates the proxy progress bar.

Parameters **count** (*int*) – The new amount of progress.

class pyphysim.progressbar.progressbar.**ProgressbarMultiProcessServer**(*progresschar: str = '*', message: str = '', sleep_time: float = 1.0, filename: Optional[str] = None, style: str = 'text2'*)

Bases: *pyphysim.progressbar.progressbar.ProgressbarDistributedServerBase*

Class that prints a representation of the current progress of multiple process as text.

While the *ProgressbarText* class only tracks the progress of a single process, the *ProgressbarMultiProcessServer* class can track the joint progress of multiple processes. This may be used, for instance, when you parallelize some task using the multiprocessing module.

Using the ProgressbarMultiProcessServer class requires a little more work than using the ProgressbarText class, as it is described in the following:

1. First you create an object of the ProgressbarMultiProcessServer class as usual. However, differently from the ProgressbarText class you don't pass the *finalcount* value to the progressbar yet.
2. After that, for each process to be tracked, call the *register_client_and_get_proxy_progressbar()* method passing the number equivalent to full progress for **that process**. This function returns a "proxy progressbar" that behaves like a regular ProgressbarText. Pass that proxy progressbar as an argument to that process so that it can call its "progress" method. Each process that calls the "progress" method of the received proxy progressbar will actually update the progress of the main ProgressbarMultiProcessServer object.
3. Start all the processes and call the start_updater method of ProgressbarMultiProcessServer object so that the bar is updated by the different processes.
4. After joining all the process (all work is finished) call the stop_updater method of the ProgressbarMultiProcessServer object.

Parameters

- **progresschar** (*str*) – Character used in the progressbar.
- **message** (*str*) – Message writen in the progressbar.
- **sleep_time** (*float*) – Time between progressbar updates (in seconds).
- **filename** (*str*) – If filename is None (default) then progress will be output to sys.stdout. If it is not None then the progress will be output to a file with name *filename*. This is usually useful for debugging and testing purposes.

Examples

```
import multiprocessing
# Create a ProgressbarMultiProcessServer object
pb = ProgressbarMultiProcessServer(message="some message")
# Creates a proxy progressbar for one process passing the value
# corresponding to 100% progress for the first process
proxybar1 = pb.register_client_and_get_proxy_progressbar(60)
# Creates a proxy progressbar for another process
proxybar2 = pb.register_client_and_get_proxy_progressbar(80)
# Create the first process passing the first proxy progressbar as
# an argument
p1 = multiprocessing.Process(target=some_function, args=[proxybar1])
# Creates another process
p2 = multiprocessing.Process(target=some_function, args=[proxybar2])
# Start both processes
p1.start()
p2.start()
# Call the start_updater method of the ProgressbarMultiProcessServer
pb.start_updater()
# Joint the process and then call the stop_updater method of the
# ProgressbarMultiProcessServer
p1.join()
p2.join()
pb.stop_updater()
```

_update_client_data_list() → None

This method process the communication between the client and the server.

register_client_and_get_proxy_progressbar(*total_count: int*) → *pyphysim.progressbar.progressbar.ProgressbarMultiProcessClient*

Register a new "client" for the progressbar and returns a new proxy progressbar that the client can use to update its progress by calling the *progress* method of this proxy progressbar.

The function whose process is tracked by the ProgressbarMultiProcessServer must must call the *progress* method of the returned ProgressbarMultiProcessClient object with the current count. This is a little less intrusive regarding the tracked function.

Parameters **total_count** (*int*) – Total count that will be equivalent to 100% for function.

Returns **obj** – The proxy progressbar.

Return type *ProgressbarMultiProcessClient*

class pyphysim.progressbar.progressbar.**ProgressbarText**(*finalcount: int*, *progresschar: str = '*'*, *message: str = ''*, *output: Any = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *pyphysim.progressbar.progressbar.ProgressbarTextBase*

Class that prints a representation of the current progress as text.

You can set the final count for the progressbar, the character that will be printed to represent progress and a small message indicating what the progress is related to.

In order to use this class, create an object outsize a loop and inside the loop call the *progress* function with the number corresponding to the progress (between 0 and finalcount). Each time the *progress* function is called a

number of characters will be printed to show the progress. Note that the number of printed characters correspond is equivalent to the progress minus what was already printed.

See also:

DummyProgressbar

Examples

>> pb = ProgressbarText(100, 'o', "Hello Simulation") >> pb.start() ————————- Hello Simulation ————————1

1 2 3 4 5 6 7 8 9 0

—-0—-0—-0—-0—-0—-0—-0—-0—-0—-0 >> pb.progress(20) ooooooooo >> pb.progress(40) oooooooooooooooooo >> pb.progress(50) ooooooooooooooooooooooo >> pb.progress(100) ooooooooooooooooooooooooooooooooooooooooooooooooooo

_ProgressbarText__get_initialization_bartitle() → str
Get the progressbar title.

The title is the first line of the progressbar initialization message.

The bar title is something like the line below

————————- % Progress ————————1

when there is no message.

Returns The bar title.

Return type str

Notes

This method is only a helper method called in the *_perform_initialization* method.

_ProgressbarText__get_initialization_markers() → Tuple[str, str]
The initialization markers 'mark' the current progress in the progressbar that will appear below it.

Returns A tuple containing the 'two lines' with the progress markers. That is, (marker_line1, marker_line2)

Return type Tuple[str, str]

Notes

This method is only a helper method called in the *_perform_initialization* method.

_perform_initialization() → None
Perform any initializations for the progressbar.

This method should be implemented in sub-classes if any initialization code should be run.

_update_iteration(*count: int*) → None
Update the progressbar according with the new *count*.

Parameters **count** (*int*) – The current count to be represented in the progressbar. The progressbar represents this count as a percent value of self.finalcount

class pyphysim.progressbar.progressbar.**ProgressbarText2**(*finalcount: int*, *progresschar: str = '*'*, *message: str = ''*, *output: Any = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *pyphysim.progressbar.progressbar.ProgressbarTextBase*

Class that prints a representation of the current progress as text.

You can set the final count for the progressbar, the character that will be printed to represent progress and a small message indicating what the progress is related to.

In order to use this class, create an object outsize a loop and inside the loop call the *progress* function with the number corresponding to the progress (between 0 and finalcount). Each time the *progress* function is called a number of characters will be printed to show the progress. Note that the number of printed characters correspond is equivalent to the progress minus what was already printed.

Parameters

- **finalcount** (*int*) – The total amount that corresponds to 100%. Each time the progress method is called with a number that number is added with the current amount in the progressbar. When the amount becomes equal to *finalcount* the bar will be 100% complete.
- **progresschar** (*str, optional (default to '*')*) – The character used to represent progress.
- **message** (*str, optional*) – A message to be shown in the right of the progressbar. If this message contains "{elapsed_time}" it will be replaced by the elapsed time.
- **output** (*File like object*) – Object with a 'write' method, which controls where the progress-bar will be printed. By default sys.stdout is used, which means that the progress will be printed in the standard output.

_update_iteration(*count: int*) → None

Update the self.prog_bar member variable according with the new *count*.

Parameters **count** (*int*) – The current count to be represented in the progressbar. The progressbar represents this count as a percent value of self.finalcount

class pyphysim.progressbar.progressbar.**ProgressbarText3**(*finalcount: int*, *progresschar: str = '*'*, *message: str = ''*, *output: Any = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: *pyphysim.progressbar.progressbar.ProgressbarTextBase*

Class that prints a representation of the current progress as text.

You can set the final count for the progressbar, the character that will be printed to represent progress and a small message indicating what the progress is related to.

In order to use this class, create an object outsize a loop and inside the loop call the *progress* function with the number corresponding to the progress (between 0 and finalcount). Each time the *progress* function is called a number of characters will be printed to show the progress. Note that the number of printed characters correspond is equivalent to the progress minus what was already printed.

Parameters

finalcount [int] The total amount that corresponds to 100%. Each time the progress method is called with a number that number is added with the current amount in the progressbar. When the amount becomes equal to *finalcount* the bar will be 100% complete.

progresschar [str, optional (default to '*')] The character used to represent progress.

message [str, optional] A message to be shown in the progressbar.

output [File like object] Object with a 'write' method, which controls where the progress-bar will be printed. By default sys.stdout is used, which means that the progress will be printed in the standard output.

_update_iteration(*count: int*) → None
Update the self.prog_bar member variable according with the new *count*.

Parameters **count** (*int*) – The current count to be represented in the progressbar. The progressbar represents this count as a percent value of self.finalcount

class pyphysim.progressbar.progressbar.**ProgressbarTextBase**(*finalcount: int*, *progresschar: str = '*'*, *message: str = ''*, *output: Any = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)
Bases: *pyphysim.progressbar.progressbar.ProgressBarBase*

Base class for Text progressbars.

Parameters

- **finalcount** (*int*) – The total amount that corresponds to 100%. Each time the progress method is called with a number that number is added with the current amount in the progressbar. When the amount becomes equal to *finalcount* the bar will be 100% complete.
- **progresschar** (*str, optional*) – The character used to represent progress.
- **message** (*str, optional*) – A message to be shown in the top of the progressbar.
- **output** (*File like object*) – Object with a 'write' method, which controls where the progress-bar will be printed. By default sys.stdout is used, which means that the progress will be printed in the standard output.

Notes

Derived classes must implement at least *_update_iteration* and this method should update the *prog_bar* member variable with the text representation of the progress.

_display_current_progress() → None
Refresh the progress representation.

All text progressbars should implement the *_update_iteration* to update the *prog_bar* member variable with the text representation of the progressbar.

This method is responsible to sending this text representation to the output.

_get_percentage_representation(*percent: float*, *central_message: str = '{percent}%'*, *left_side: str = '['*, *right_side: str = ']'*) → str
Get the percent representation as a string suitable to the text progressbar.

Parameters

- **percent** (*float*) – The percentage to be represented.
- **central_message** (*str*) – A message that will be in the middle of the percentage bar. If there is the label '{percent}' in the central_message it will be replaced by the percentage. If there is the label '{elapsed_time}' in the central_message it will be replaced by the elapsed time. Note that this message should be very small, since it hides the progresschars.
- **left_side** (*str*) – The left side of the bar.
- **right_side** (*str*) – The right side of the bar.

Returns **representation** – A string with the representation of the percentage.

Return type str

_maybe_delete_output_file() → None
Delete the output file (if there is any) when delete_progress_file_after_completion is set to True.

_perform_finalizations() → None
Perform any finalization (cleanings) after the progressbar stops.

property width
Get method for the width property.

class pyphysim.progressbar.progressbar.**ProgressbarZMQClient**(*client_id: int*, *ip: str*, *port: int*, *finalcount: int*)
Bases: *pyphysim.progressbar.progressbar.ProgressbarDistributedClientBase*

Proxy progressbar that behaves like a ProgressbarText object, but is actually updating a ProgressbarZMQServer progressbar.

The basic idea is that this proxy progressbar has the "progress" method similar to the standard ProgressbarText class. However, when this method is called it will update a value that will be read by a ProgressbarZMQServer object instead.

Parameters

- **client_id** (*ClientID*) – The client ID.
- **ip** (*IPAddress*) – A string representing the IP address of the server.
- **port** (*PortNumber*) – The port number used by the server.

_perform_initialization() → None
Creates the "push socket", connects it to the socket of the main progressbar and then updates the progress.

This function will be called only in the first time the "progress" method is called. Subsequent calls to "progress" will actually calls the "_progress" method.

Parameters **count** (*int*) – The new amount of progress.

_update_iteration(*count: int*) → None
Updates the proxy progress bar.

Parameters **count** (*int*) – The new amount of progress.

class pyphysim.progressbar.progressbar.**ProgressbarZMQServer**(*progresschar: str = '*'*, *message: str = ''*, *sleep_time: float = 1.0*, *filename: Optional[str] = None*, *ip: str = 'localhost'*, *port: int = 7396*, *style: str = 'text2'*)

Bases: *pyphysim.progressbar.progressbar.ProgressbarDistributedServerBase*

Distributed "server" progressbar using ZMQ sockets.

In order to track the progress of distributed computations two classes are required, one that acts as a central point and is responsible to actually show the progress (the server), and other class that acts as a proxy (the client) and is responsible to sending the current progress to the server. There will be one object of the "server class" and one or more objects of the "client class", each one tracking the progress of one of the distributed computations.

This class acts like the server. It creates a ZMQ socket which expects (string) messages in the form "client_id:current_progress", where the client_id is the ID of one client progressbar previously registered with the register_client_and_get_proxy_progressbar method while the current_progress is a "number" with the current progress of that client.

Note that the client proxybar for this class is implemented in the ProgressbarZMQClient class.

Parameters

- **progresschar** (*str*) – Character used in the progressbar.
- **message** (*str*) – Message writen in the progressbar.
- **sleep_time** (*float*) – Time between progressbar updates (in seconds).
- **filename** (*str*) – If filename is None (default) then progress will be output to sys.stdout. If it is not None then the progress will be output to a file with name *filename*. This is usually useful for debugging and testing purposes.
- **ip** (*IPAddress*) – An string representing the address of the server socket. Ex: '192.168.0.117', 'localhost', etc.
- **port** (*PortNumber*) – The port to bind the socket.

_parse_progress_message(*message: str*) → None

Parse the message sent from the client proxy progressbars.

The messages sent from the proxy progressbars are in the form 'client_id:current_count'. We need to set the element of index "client_id" in self._client_data_list to the value of "current_count". This method will simply parse the message and perform this operation.

Parameters **message** (*str*) – A string in the form 'client_id:current_count'.

_update_client_data_list() → None

This method process the communication between the client and the server.

This method will read the received messages in the socket which were sent by the clients (ProgressbarZMQClient objects) and update self._client_data_list variable accordingly. The messages are in the form "client_id:current_progress", which is parsed by the _parse_progress_message method.

Notes

This method is called inside a loop in the _update_progress method.

_update_progress(*filename: Optional[str] = None*, *start_delay: float = 0.0*) → None

Collects the progress from each registered proxy progressbar and updates the actual visible progressbar.

Parameters

- **filename** (*str*) – Name of a file where the data will be written to. If this is None then all progress will be printed in the standard output (default)

- **start_delay** (*float*) – Delay in seconds before starting the progressbar. During this time it is still possible to register new clients and the progressbar will only be shown after this delay.

Notes

We re-implement it here only to create the ZMQ socket. After that we call the base class implementation of _update_progress method. Note that the _update_progress method in the base class calls the _update_client_data_list and we indeed re-implement this method in this class and use the socket created here in that implementation.

property ip
Get method for the ip property.

Returns The string representing the address of the server socket.

Return type str

property num_clients
Number of registered clients

property port
Get method for the port property.

Returns The port used.

Return type int

register_client_and_get_proxy_progressbar(*total_count: int*) → *pyphysim.progressbar.progressbar.ProgressbarZMQClient*
Register a new client progressbar and return a proxy to it.

Parameters **total_count** (*int*) – The total count for the client we are registering.

Returns The proxy progressbar.

Return type *ProgressbarZMQClient*

Module contents

pyphysim.reference_signals package

Submodules

pyphysim.reference_signals.channel_estimation module

Module with channel estimation implementations based on the reference signals in this package.

class pyphysim.reference_signals.channel_estimation.**CazacBasedChannelEstimator**(*ue_ref_seq: Union[pyphysim.reference_signals.cazac_sequences.CazacSequence, numpy.ndarray], size_multiplier: int = 2*)

Bases: object

Estimated the (uplink) channel based on CAZAC (Constant Amplitude Zero AutoCorrelation) reference sequences sent by one user (either SRS or DMRS).

The estimation is performed according to the paper [Bertrand2011], where the received signal in the FREQUENCY DOMAIN is used by the estimator.

Note that for SRS sequences usually a comb pattern is employed such that only half of the subcarriers is used to send pilot symbols. Therefore, an FFT in the during the estimation will effectively interpolate for the other subcarriers. This is controlled by the *size_multiplier* argument (default is 2 to accommodate comb pattern). If all subcarriers are used to send pilot symbols then set *size_multiplier* to 1.

Parameters

- **ue_ref_seq** (*SrsUeSequence | DmrsUeSequence | np.ndarray*) – The reference signal sequence.
- **size_multiplier** (*int, optional*) – Multiplication factor for the FFT to get the actual channel size. When using the comb pattern for SRS this should be 2 (default value), but for DMRS, which does not employ the comb pattern, this should be set to 1.

Notes

estimate_channel_freq_domain(*received_signal: numpy.ndarray*, *num_taps_to_keep: int*) → numpy.ndarray

Estimate the channel based on the received signal.

Parameters

- **received_signal** (*np.ndarray*) – The received reference signal after being transmitted through the channel (in the frequency domain). If this is a 2D numpy array the first dimensions is assumed to be "receive antennas" while the second dimension are the received sequence elements. The number of elements in the received signal (per antenna) is equal to the channel size (number of subcarriers) divided by *size_multiplier*.
- **num_taps_to_keep** (*int*) – Number of taps (in delay domain) to keep. All taps from 0 to *num_taps_to_keep*-1 will be kept and all other taps will be zeroed before applying the FFT to get the channel response in the frequency domain.

Returns **freq_response** – The channel frequency response. Note that for SRS sequences this will have twice as many elements as the sent SRS signal, since the SRS signal is sent every other subcarrier.

Return type np.ndarray

property ue_ref_seq

Get the sequence of the UE.

class pyphysim.reference_signals.channel_estimation.**CazacBasedWithOCCChannelEstimator**(*ue_ref* py-physin

Bases: *pyphysim.reference_signals.channel_estimation.CazacBasedChannelEstimator*

Estimated the (uplink) channel based on CAZAC (Constant Amplitude Zero AutoCorrelation) reference sequences sent by one user including the Orthogonal Cover Code (OCC).

With OCC the user will send reference signal in multiple time slots, in each slot multiplied with the respective OCC sequence element.

Parameters **ue_ref_seq** (DmrsUeSequence) – The reference signal sequence.

property cover_code
Get the cover code of the UE.

estimate_channel_freq_domain(*received_signal: numpy.ndarray*, *num_taps_to_keep: int*, *extra_dimension: bool = True*) → numpy.ndarray
Estimate the channel based on the received signal with cover codes.

Parameters

- **received_signal** (*np.ndarray*) – The received reference signal after being transmitted through the channel (in the frequency domain).

  Dimension: Depend if there are multiple receive antennas and if *extra_dimension* is True or False. Let Nr be the number of receive antennas, Ne be the number of reference signal elements (reference signal size without cover code) and Nc be the cover code size. The dimension of *received_signal* must match the table below.

/	extra_dimension: True	extra_dimension: False
Single Antenna	Nc x Ne (2D)	Ne * Nc (1D)
Multiple Antennas	Nr x Nc x Ne (3D)	Nr x (Ne * Nc) (2D)

- **num_taps_to_keep** (*int*) – Number of taps (in delay domain) to keep. All taps from 0 to *num_taps_to_keep*-1 will be kept and all other taps will be zeroed before applying the FFT to get the channel response in the frequency domain.
- **extra_dimension** (*bool*) – If True then the should be an extra dimension in *received_signal* corresponding to the cover code dimension. If False then the cover code is included in the dimension of the reference signal elements.

Returns **freq_response** – The channel frequency response.

Return type np.ndarray

pyphysim.reference_signals.dmrs module

Module with Sounding Reference Signal (SRS) related functions

class pyphysim.reference_signals.dmrs.**DmrsUeSequence**(*root_seq: pyphysim.reference_signals.root_sequence.RootSequence*, *n_cs: int*, *cover_code: Optional[numpy.ndarray] = None*, *normalize: bool = False*)
Bases: pyphysim.reference_signals.srs.UeSequence

DMRS sequence of a single user.

Parameters

- **root_seq** (RootSequence) – The DMRS root sequence of the base station the user is associated to. This should be an object of the RootSequence class.
- **n_cs** (*int*) – The shift index of the user. This can be an integer from 0 to 11.
- **cover_code** (*np.ndarray, optional*) – Cover Code used by the UE. As an example, consider the cover code *np.array([1, -1])*. In that case, if the regular DMRS sequence (without the cover code) is *seq*, than the actual DMRS sequence with cover code will be a 2D numpy array equivalent with *seq_occ[0]==seq* and *seq_occ[1]==-seq*.
- **normalize** (*bool*) – True if the reference signal should be normalized. False otherwise.

property cover_code
Return the cover code.

property size
Return the size of the reference signal sequence.

Returns **size** – The size of the user's reference signal sequence.

Return type int

pyphysim.reference_signals.dmrs.**get_dmrs_seq**(*root_seq: numpy.ndarray*, *n_cs: int*) → numpy.ndarray
Get the shifted root sequence suitable as the DMRS sequence of a user.

Parameters

- **root_seq** (*np.ndarray*) – The root sequence to shift.
- **n_cs** (*int*) – The desired cyclic shift number. This should be an integer from 0 to 11, where 0 will just return the base sequence, 1 gives the first shift, and so on.

Returns The shifted root sequence.

Return type np.ndarray

See also:

zadoffchu.get_shifted_root_seq(), *srs.get_srs_seq()*

pyphysim.reference_signals.root_sequence module

Module with Sounding Reference Signal (SRS) related functions

class pyphysim.reference_signals.root_sequence.**RootSequence**(*root_index: int*, *size: Optional[int] = None*, *Nzc: Optional[int] = None*)
Bases: object

Class representing the root sequence of the reference signals.

The root sequence is generated using two possible formulas, one used for a sequence size smaller than $3M_{sc}^{RS}$ and another for sequence size equal to or greater than $3M_{sc}^{RS}$, where $3M_{sc}^{RS}$ is the number of subcarriers in a PRB (12 subcarriers).

Parameters

- **root_index** (*int*) – The SRS root sequence index.
- **size** (*int*) – The size of the extended Zadoff-Chu sequence. If None then the sequence will not be extended and will thus have a size equal to Nzc.
- **Nzc** (*int*) – The size of the Zadoff-Chu sequence (without any extension). If not provided then the largest prime number lower than or equal to *size* will be used.

property Nzc
Get the size of the Zadoff-Chu sequence (without any extension).

Returns The value of the Nzc property.

Return type int

static _get_largest_prime_lower_than_number(*seq_size: int*) → int
Get the largest prime number lower than *seq_size*.

Parameters **seq_size** (*int*) – The sequence size.

Returns The largest prime number lower than *seq_size*.

Return type int

conj() → numpy.ndarray
Return the conjugate of the root sequence as a numpy array.

Returns The conjugate of the root sequence.

Return type np.ndarray

conjugate() → numpy.ndarray
Return the conjugate of the root sequence as a numpy array.

Returns The conjugate of the root sequence.

Return type np.ndarray

property index
Return the SRS root sequence index.

Returns The root sequence index.

Return type int

n_sc_PRB = 12

seq_array() → numpy.ndarray
Get the extended Zadoff-Chu root sequence as a numpy array.

Returns **seq** – The extended Zadoff-Chu sequence

Return type np.ndarray

property size
Return the size (with extension) of the sequence.

If the sequence is not extended than *size()* will return the same as *Nzc*.

Returns **size** – The size of the extended Zadoff-Chu sequence.

Return type int

Examples

```
>>> seq1 = RootSequence(root_index=25, Nzc=139)
>>> seq1.size
139
>>> seq1 = RootSequence(root_index=25, Nzc=139, size=150)
>>> seq1.size
150
```

pyphysim.reference_signals.srs module

Module with Sounding Reference Signal (SRS) related functions

class pyphysim.reference_signals.srs.**SrsUeSequence**(*root_seq:* pyphysim.reference_signals.root_sequence.RootSequence, *n_cs:* *int*, *normalize:* *bool* = *False*)

Bases: pyphysim.reference_signals.srs.UeSequence

SRS sequence of a single user.

Parameters

- **root_seq** (RootSequence) – The SRS root sequence of the base station the user is associated to. This should be an object of the RootSequence class.
- **n_cs** (*int*) – The shift index of the user. This can be an integer from 0 to 7.
- **normalize** (*bool*) – True if the reference signal should be normalized. False otherwise.

pyphysim.reference_signals.srs.**get_srs_seq**(*root_seq:* *numpy.ndarray*, *n_cs:* *int*) → numpy.ndarray

Get the shifted root sequence suitable as the SRS sequence of a user.

Parameters

- **root_seq** (*np.ndarray*) – The root sequence to shift. This is a complex numpy array.
- **n_cs** (*int*) – The desired cyclic shift number. This should be an integer from 0 to 7, where 0 will just return the base sequence, 1 gives the first shift, and so on.

Returns The shifted root sequence.

Return type np.ndarray

See also:

get_shifted_root_seq(), *dmrs.get_dmrs_seq()*

pyphysim.reference_signals.zadoffchu module

Module containing Zadoff-chu related functions.

pyphysim.reference_signals.zadoffchu.**calcBaseZC**(*Nzc:* *int*, *u:* *int*, *q:* *complex* = *0*) → numpy.ndarray

Calculate the root sequence of Zadoff-Chu sequences.

Parameters

- **Nzc** (*int*) – The size of the root Zadoff-Chu sequence.
- **u** (*int*) – The root sequence index.
- **q** (*complex*) – Any complex number. Usually this is just zero.

Returns **a_u** – The root Zadoff-Chu sequence.

Return type np.ndarray

pyphysim.reference_signals.zadoffchu.**get_extended_ZF**(*root_seq:* *numpy.ndarray*, *size:* *int*) → numpy.ndarray

Cyclic Extend the Zadoff-Chu root sequence to have size equal to *size*.

Parameters

- **root_seq** (*np.ndarray*) – The root Zadoff-Chu sequence. This is a complex numpy array.
- **size** (*int*) – The size that the sequence should be extended to.

Returns **output** – The extended root sequence.

Return type np.ndarray

Examples

```
>>> root_seq = np.array([1, 2, 3, 4, 5])
>>> get_extended_ZF(root_seq, 8)
array([1, 2, 3, 4, 5, 1, 2, 3])
```

pyphysim.reference_signals.zadoffchu.**get_shifted_root_seq**(*root_seq: numpy.ndarray*, *n_cs: int*, *denominator: int*) → numpy.ndarray

Get the shifted root sequence suitable as the SRS sequence or the DMRS sequence of a user (depend on the *denominator* parameter).

Parameters

- **root_seq** (*np.ndarray*) – The root sequence to be shifted. This is a complex numpy array.
- **n_cs** (*int*) – The desired cyclic shift number. This should be an integer from 0 to *denominator*-1, where 0 will just return the base sequence, 1 gives the first shift, and so on.
- **denominator** (*int*) – The denominator in the cyclic shift formula. This should be 8 for SRS and 12 for DMRS.

Returns The shifted root sequence (a complex numpy array).

Return type np.ndarray

See also:

`get_srs_seq()`, `get_dmrs_seq()`

Module contents

Package for Uplink Reference Signals related functionality.

pyphysim.simulations package

Submodules

pyphysim.simulations.configobjvalidation module

Module implementing validation functions to define "specs" for configobj validation.

This module is not intended to be used directly. The functions defined here are used in the other modules in the *pyphysim.simulations* package.

pyphysim.simulations.configobjvalidation.**integer_numpy_array_check**(*value: str*, *min: Optional[int] = None*, *max: Optional[int] = None*) → List[int]

Parse and validate *value* as a numpy array (of integers).

Value can be either a single number, a range expression in the form of min:max or min:step:max, or even a list containing numbers and range expressions.

Parameters

- **value** (*str*) – The string to be converted. This can be either a single number, a range expression in the form of min:max or min:step:max, or even a list containing numbers and range expressions.
- **min** (*int*) – The minimum allowed value. If the converted value is (or have) lower than *min* then the VdtValueTooSmallError exception will be raised.
- **max** (*int*) – The maximum allowed value. If the converted value is (or have) greater than *man* then the VdtValueTooSmallError exception will be raised.

Returns The parsed numpy array.

Return type List[int]

Notes

You can either separate the values with commas or spaces (any comma will have the same effect as a space). However, if you separate with spaces the values should be brackets, while if you separate with commands there should be no brackets.

>> max_iter = 5,10:20 >> max_iter = [0 5 10:20]

pyphysim.simulations.configobjvalidation.**integer_scalar_or_integer_numpy_array_check**(*value: str*, *min: Optional[int] = None*, *max: Optional[int] = None*) → Union[int, List[int]]

Parse and validate *value* as an integer number if possible and, if not, parse it as a numpy array (of integers).

Value can be either a single number, a range expression in the form of min:max or min:step:max, or even a list containing numbers and range expressions. The difference regarding the *integer_numpy_array_check* function is that if value is a single number it will be parsed as a single integer value, instead of being parsed as an integer numpy array with a single element.

Parameters

- **value** (*str*) – The string to be converted. This can be either a single number, a range expression in the form of min:max or min:step:max, or even a list containing numbers and range expressions.
- **min** (*int*) – The minimum allowed value. If the converted value is (or have) lower than *min* then the VdtValueTooSmallError exception will be raised.
- **max** (*int*) – The maximum allowed value. If the converted value is (or have) greater than *man* then the VdtValueTooSmallError exception will be raised.

Returns The parsed numpy array.

Return type int | List[int]

Notes

You can either separate the values with commas or spaces (any comma will have the same effect as a space). However, if you separate with spaces the values should be brackets, while if you separate with commands there should be no brackets.

>> max_iter = 5,10:20 >> max_iter = [0 5 10:20]

pyphysim.simulations.configobjvalidation.**real_numpy_array_check**(*value: str*, *min: Optional[int] = None*, *max: Optional[int] = None*)

Parse and validate *value* as a numpy array (of floats).

Value can be either a single number, a range expression in the form of min:max or min:step:max, or even a list containing numbers and range expressions.

Parameters

- **value** (*str*) – The string to be converted. This can be either a single number, a range expression in the form of min:max or min:step:max, or even a list containing numbers and range expressions.
- **min** (*int*) – The minimum allowed value. If the converted value is (or have) lower than *min* then the VdtValueTooSmallError exception will be raised.
- **max** (*int*) – The maximum allowed value. If the converted value is (or have) greater than *man* then the VdtValueTooSmallError exception will be raised.

Returns The parsed numpy array.

Return type List[float]

Notes

You can either separate the values with commas or spaces (any comma will have the same effect as a space). However, if you separate with spaces the values should be in brackets, while if you separate with commands there should be no brackets.

>> SNR = 0,5,10:20 >> SNR = [0 5 10:20]

pyphysim.simulations.configobjvalidation.**real_scalar_or_real_numpy_array_check**(*value: str*, *min=None*, *max=None*)

Parse and validate *value* as a float number if possible and, if not, parse it as a numpy array (of floats).

Value can be either a single number, a range expression in the form of min:max or min:step:max, or even a list containing numbers and range expressions. The difference regarding the *real_numpy_array_check* function is that if value is a single number it will be parsed as a single float value, instead of being parsed as a real numpy array with a single element.

Parameters

- **value** (*str | list[str]*) – The string to be converted. This can be either a single number, a range expression in the form of min:max or min:step:max, or even a list containing numbers and range expressions.
- **min** (*int | float*) – The minimum allowed value. If the converted value is (or have) lower than *min* then the VdtValueTooSmallError exception will be raised.
- **max** (*int | float*) – The maximum allowed value. If the converted value is (or have) greater than *man* then the VdtValueTooSmallError exception will be raised.

Returns The parsed numpy array.

Return type float | List[float]

Notes

You can either separate the values with commas or spaces (any comma will have the same effect as a space). However, if you separate with spaces the values should be in brackets, while if you separate with commands there should be no brackets.

>> SNR = 0,5,10:20 >> SNR = [0 5 10:20]

pyphysim.simulations.parameters module

Module containing simulation parameter classes.

class pyphysim.simulations.parameters.**SimulationParameters**

Bases: *pyphysim.util.serialize.JsonSerializable*

Class to store the simulation parameters.

A SimulationParameters object acts as a container for all simulation parameters. To add a new parameter to the object just call the *add()* method passing the name and the value of the parameter. The value can be anything as long as the *SimulationRunner._run_simulation()* method can understand it.

Alternatively, you can create a SimulationParameters object with all the parameters with the static method *create()*, which receives a dictionary with the parameter names as keys.

Parameters can be marked to be "unpacked", as long as they are iterable, by calling the *set_unpack_parameter()* method. Different simulations will be performed for every combination of parameters marked to be unpacked, with the other parameters kept constant.

Examples

- Create a new empty SimulationParameters object and add the individual parameters to it by calling its *add()* method.

```
params = SimulationParameters()
params.add('p1', [1,2,3])
params.add('p2', ['a','b'])
params.add('p3', 15)
```

- Creating a new SimulationParameters object with the static *create()* function.

```
p = {'p1':[1,2,3], 'p2':['a','b'],'p3':15}
params=SimulationParameters.create(p)
params.set_unpack_parameter('p1')
params.set_unpack_parameter('p2')
```

- We can then set some of the parameters to be unpacked with

```
params.set_unpack_parameter('p1')
```

See also:

SimulationResults Class to store simulation results.

SimulationRunner Base class to implement Monte Carlo simulations.

static _create(*params_dict*, *unpack_index=- 1*, *original_sim_params=None*)
Creates a new SimulationParameters object.

This static method provides a different way to create a SimulationParameters object, already containing the parameters in the *params_dict* dictionary.

Parameters

- **params_dict** (*dict*) – Dictionary containing the parameters. Each dictionary key corresponds to a parameter's name, while the dictionary value corresponds to the actual parameter value..
- **unpack_index** (*int*) – Index of the created SimulationParameters object when it is part of the unpacked variations of another SimulationParameters object. See *get_unpacked_params_list()*.
- **original_sim_params** (SimulationParameters) – The original SimulationParameters object from which the SimulationParameters object that will be created by this method came from.

Returns **sim_params** – The corresponding SimulationParameters object.

Return type *SimulationParameters*

static _from_dict(*d*)
Create a new SimulationParameters object from a dictionary.

Parameters **d** (*dict*) – The dictionary with the data.

Returns The new SimulationParameters object.

Return type *SimulationParameters*

_to_dict()
Convert the SimulationParameters object to a dictionary for easier further serialization.

add(*name*, *value*)
Adds a new parameter to the SimulationParameters object.

If there is already a parameter with the same name it will be replaced.

Parameters

- **name** (*str*) – Name of the parameter.
- **value** (*anything*) – Value of the parameter.

static create(*params_dict*)
Creates a new SimulationParameters object.

This static method provides a different way to create a SimulationParameters object, already containing the parameters in the *params_dict* dictionary.

Parameters **params_dict** (*dict*) – Dictionary containing the parameters. Each dictionary key corresponds to a parameter's name, while the dictionary value corresponds to the actual parameter value..

Returns **sim_params** – The corresponding SimulationParameters object.

Return type *SimulationParameters*

property fixed_parameters
Names of the parameters which are NOT marked to be unpacked.

Returns List with the names of the fixed parameter.

Return type list[str]

get_num_unpacked_variations()
Get the number of variations when the parameters are unpacked.

If no parameter was marked to be unpacked, then return 1.

If this SimulationParameters object is actually a 'unpacked variation' of another SimulationParameters object, return the number of variations of the parent SimulationParameters object instead.

Returns **num** – The number of variations when the parameters are unpacked.

Return type int

get_pack_indexes(*fixed_params_dict=None*)
When you call the function get_unpacked_params_list you get a list of SimulationParameters objects corresponding to all combinations of the parameters. The function get_pack_indexes allows you to provided all parameters marked to be unpacked except one, and returns the indexes of the list returned by get_unpacked_params_list that you want.

Parameters **fixed_params_dict** (*dict[str, anything]*) – A dictionary with the name of the fixed parameters as keys and the fixed value as value.

Returns **indexes** – The desired indexes (1D numpy array or an integer).

Return type np.ndarray

Examples

Suppose we have

```
>>> p={'p1':[1,2,3], 'p2':['a','b'],'p3':15}
>>> params=SimulationParameters.create(p)
>>> params.set_unpack_parameter('p1')
>>> params.set_unpack_parameter('p2')
```

If we call params.get_unpacked_params_list we will get a list of six SimulationParameters objects, one for each combination of the values of p1 and p2. That is,

```
>>> len(params.get_unpacked_params_list())
6
```

Likewise, in the simulation the *SimulationRunner* object will return a list of results in the order corresponding to the order of the list of parameters. The get_pack_indexes is used to get the index of the results corresponding to a specific configuration of parameters. Suppose now you want the results when 'p2' is varying, but with the other parameters fixed to some specific value. For this create a dictionary specifying all parameters except 'p2' and call get_pack_indexes with this dictionary. You will get an array of indexes that can be used in the results list to get the desired results. For instance

```
>>> fixed={'p1':3,'p3':15}
>>> indexes = params.get_pack_indexes(fixed)
>>> index0 = indexes[0]
>>> index1 = indexes[1]
>>> unpacked_list = params.get_unpacked_params_list()
>>> unpacked_list[index0]['p1']
3
>>> unpacked_list[index0]['p3']
15
>>> unpacked_list[index1]['p1']
3
>>> unpacked_list[index1]['p3']
15
```

get_unpacked_params_list()

Get a list of SimulationParameters objects, each one corresponding to a possible combination of (unpacked) parameters.

Returns A list of SimulationParameters objects.

Return type list[*SimulationParameters*]

Examples

Suppose you have a SimulationParameters object with the parameters 'a', 'b', 'c' and 'd' as below

```
>>> simparams = SimulationParameters()
>>> simparams.add('a', 1)
>>> simparams.add('b', 2)
>>> simparams.add('c', [3, 4])
>>> simparams.add('d', [5, 6])
```

and the parameters 'c' and 'd' were set to be unpacked.

```
>>> simparams.set_unpack_parameter('c')
>>> simparams.set_unpack_parameter('d')
```

Then get_unpacked_params_list would return a list of four SimulationParameters objects as below

```
>>> len(simparams.get_unpacked_params_list())
4
```

That is

```
[{'a': 1, 'c': 3, 'b': 2, 'd': 5},
 {'a': 1, 'c': 3, 'b': 2, 'd': 6},
 {'a': 1, 'c': 4, 'b': 2, 'd': 5},
 {'a': 1, 'c': 4, 'b': 2, 'd': 6}]
```

static load_from_config_file(*filename*, *spec=None*, *save_parsed_file=False*)

Load the SimulationParameters from a config file using the configobj module.

If the config file has a parameter called `unpacked_parameters`, which should be a list of strings with the names of other parameters, then these parameters will be set to be unpacked.

Parameters

- **filename** (*str*) – Name of the file from where the results will be loaded.
- **spec** (*list[str], optional*) – A list of strings with the config spec. See "validation" in the configobj module documentation for more info.
- **save_parsed_file** (*bool*) – If *save_parsed_file* is True, then the parsed config file will be written back to disk. This will add any missing values in the config file whose default values are provided in the *spec*. This will even create the file if all default values are provided in *spec* and the file does not exist yet.

Notes

Besides the usual checks that the configobj validation has such as `integer`, `string`, `option`, etc., you can also use *real_numpy_array* for numpy float arrays. Note that when this validation function is used you can set the array in the config file in several ways such as

SNR=0,5,10,15:20

for instance.

static load_from_pickled_file(*filename*)

Load the SimulationParameters from the file 'filename' previously stored (using pickle) with *save_to_pickled_file*.

Parameters **filename** (*str*) – Name of the file from where the results will be loaded.

Returns The loaded SimulationParameters object.

Return type *SimulationParameters*

remove(*name*)

Remove the parameter with name *name* from the SimulationParameters object

Parameters **name** (*str*) – Name of the parameter to be removed.

Raises **KeyError** – If *name* is not in parameters.

save_to_pickled_file(*filename*)
Save the SimulationParameters object to the file *filename* using pickle.

Parameters **filename** (*str*) – Name of the file to save the parameters.

set_unpack_parameter(*name*, *unpack_bool=True*)
Set the unpack property of the parameter with name *name*.

The parameter *name* must be already added to the SimulationParameters object and be an iterable.

This is used in the *SimulationRunner* class.

Parameters

- **name** (*str*) – Name of the parameter to be unpacked.
- **unpack_bool** (*bool, optional*) – True activates unpacking for *name*, False deactivates it.

Raises **ValueError** – If *name* is not in parameters or is not iterable.

property unpack_index
Get method for the unpack_index property.

Returns The unpack index. Ir this SimulationParameters object is not an unpacked SimulationParameters then -1 is returned.

Return type int

property unpacked_parameters
Names of the parameters marked to be unpacked.

Returns

Return type list[str]

pyphysim.simulations.parameters.**combine_simulation_parameters**(*params1*, *params2*)
Combine two SimulationParameters objects and return a new SimulationParameters object corresponding to the union of them.

The union is only valid if both objects have the same parameters and only the values of the unpacked parameters are different.

Parameters

- **params1** (SimulationParameters) – The first SimulationParameters object.
- **params2** (SimulationParameters) – The second SimulationParameters object.

Returns **union** – The union of 'params1' and 'params2'.

Return type *SimulationParameters*

pyphysim.simulations.results module

Module containing simulation result classes.

class pyphysim.simulations.results.**Result**(*name: str*, *update_type_code: int*, *accumulate_values: bool = False*, *choice_num: Optional[int] = None*)

Bases: *pyphysim.util.serialize.JsonSerializable*

Class to store a single simulation result.

A simulation result can be anything, such as the number of errors, a string, an error rate, etc. When creating a *Result* object one needs to specify only the *name* of the stored result and the result *type*.

The different types indicate how multiple samples (from multiple iterations) of the same Result can be merged (usually to get a result with more statistical reliability). The possible values are SUMTYPE, RATIOTYPE and MISCTYPE.

In the *SUMTYPE* the new value should be added to current one in update function.

In the *RATIOTYPE* the new value should be added to current one and total should be also updated in the update function. One caveat is that rates are stored as a number (numerator) and a total (denominator) instead of as a float. For instance, if you need to store a result such as a bit error rate, then you could use the a Result with the RATIOTYPE type and when updating the result, pass the number of bit errors and the number of simulated bits.

The *MISCTYPE* type can store anything and the update will simple replace the stored value with the current value.

Parameters

- **name** (*str*) – Name of the Result.
- **update_type_code** (*int*) – Type of the result. It must be one of the elements in {Result.SUMTYPE, Result.RATIOTYPE, Result.MISCTYPE, Result.CHOICETYPE}.
- **accumulate_values** (*bool*) – If True, then the values *value* and *total* will be accumulated in the *update* (and merge) method(s). This means that the Result object will use more memory as more and more values are accumulated, but having all values sometimes is useful to perform statistical calculations. This is useful for debugging/testing.
- **choice_num** (*int*) – Number of different choices for the CHOICETYPE type. This is a required parameter for the CHOICETYPE type, but it is ignored for the other types

Examples

- Example of the SUMTYPE result.

```
>>> result1 = Result("name", Result.SUMTYPE)
>>> result1.update(13)
>>> result1.update(4)
>>> result1.get_result()
17
>>> result1.num_updates
2
>>> result1
Result -> name: 17
>>> result1.type_name
'SUMTYPE'
>>> result1.type_code
0
```

- Example of the RATIOTYPE result.

```
>>> result2 = Result("name2", Result.RATIOTYPE)
>>> result2.update(4,10)
>>> result2.update(3,4)
>>> result2.get_result()
0.5
>>> result2.type_name
'RATIOTYPE'
>>> result2.type_code
1
>>> result2_other = Result("name2", Result.RATIOTYPE)
>>> result2_other.update(3,11)
>>> result2_other.merge(result2)
>>> result2_other.get_result()
0.4
>>> result2_other.num_updates
3
>>> result2_other._value
10
>>> result2_other._total
25
>>> result2.get_result()
0.5
>>> print(result2_other)
Result -> name2: 10/25 -> 0.4
```

- Example of the MISCTYPE result.

  The MISCTYPE result 'merge' process in fact simple replaces the current stored value with the new value.

CHOICETYPE = 3

MISCTYPE = 2

RATIOTYPE = 1

SUMTYPE = 0

_all_types = {0: 'SUMTYPE', 1: 'RATIOTYPE', 2: 'MISCTYPE', 3: 'CHOICETYPE'}

static _from_dict(*d: Dict[str, Any]*) → *pyphysim.simulations.results.Result*
Convert from a dictionary to a Result object.

Parameters **d** (*dict*) – The dictionary representing the Result.

Returns The converted object.

Return type *Result*

_to_dict() → Dict[str, Any]
Convert the Result object to a dictionary representation.

Returns The dictionary representation of the object.

Return type dict

property accumulate_values_bool
Property to see if values are accumulated of not during a call to the *update* method.

static create(*name: str*, *update_type: int*, *value: Any*, *total: int = 0*, *accumulate_values: bool = False*) → *pyphysim.simulations.results.Result*
Create a Result object and update it with *value* and *total* at the same time.

Equivalent to creating the object and then calling its *update()* method.

Parameters

- **name** (*str*) – Name of the Result.
- **update_type** (*int*) – Type of the result. It must be one of the elements in {Result.SUMTYPE, Result.RATIOTYPE, Result.MISCTYPE, Result.CHOICETYPE}.
- **value** (*any*) – Value of the result.
- **total** (*any | int | float*) – Total value of the result (used only for the RATIOTYPE and CHOICETYPE). For the CHOICETYPE it is interpreted as the number of different choices if it is an integer or the current value of each choice if it is a list.
- **accumulate_values** (*bool*) – If True, then the values *value* and *total* will be accumulated in the *update* (and merge) method(s). This means that the Result object will use more memory as more and more values are accumulated, but having all values sometimes is useful to perform statistical calculations. This is useful for debugging/testing.

Returns The new Result object.

Return type *Result*

Notes

Even if accumulate_values is True the values will not be accumulated for the MISCTYPE.

See also:

update()

get_confidence_interval(*P: float = 95.0*) → numpy.ndarray
Get the confidence interval that contains the true result with a given probability *P*.

Parameters **P** (*float*) – The desired confidence (probability in %) that true value is inside the calculated interval. The possible values are described in the documentation of the *calc_confidence_interval()* function`

Returns **Interval** – Numpy (float) array with two elements.

Return type np.ndarray

See also:

calc_confidence_interval()

get_result() → Any
Get the result stored in the Result object.

Returns **results** – For the RATIOTYPE type get_result will return the *value/total*, while for the other types it will return *value*.

Return type anything, but usually a number

get_result_accumulated_totals() → List[Any]
Return the accumulated values.

Note that in case the result if of type RATIOTYPE this you probably want to call the get_result_accumulated_values function to also get the values.

get_result_accumulated_values() → List[Any]
Return the accumulated values.

Note that in case the result if of type RATIOTYPE this you probably want to call the get_result_accumulated_totals function to also get the totals.

get_result_mean() → float
Get the mean of all the updated results.

Returns The mean of the result.

Return type float

get_result_var() → float
Get the variance of all updated results.

Returns The variance of the results.

Return type float

merge(*other:* pyphysim.simulations.results.Result) → None
Merge the result from other with self.

Parameters **other** (Result) – Another Result object.

property type_code
Get the Result type.

Returns **type_code** – The returned value is a number corresponding to one of the types SUMTYPE, RATIOTYPE, MISCTYPE or CHOICETYPE.

Return type int

property type_name
Get the Result type name.

Returns **type_name** – The result type string (SUMTYPE, RATIOTYPE, MISCTYPE or CHOICETYPE).

Return type str

update(*value: Any*, *total: Optional[Any] = None*) → None
Update the current value.

Parameters

- **value** (*anything, but usually a number*) – Value to be added to (or replaced) the current value
- **total** (same type as *value*) – Value to be added to the current total (only useful for the RATIOTYPE update type)

Notes

The way how this update process depends on the Result type and is described below

- RATIOTYPE: Add "value" to current value and "total" to current total.
- SUMTYPE: Add "value" to current value. "total" is ignored.
- MISCTYPE: Replace the current value with "value".
- CHOICETYPE: Update the choice "value" and the total by 1.

See also:

create()

class pyphysim.simulations.results.**SimulationResults**
Bases: *pyphysim.util.serialize.JsonSerializable*

Store results from simulations.

This class is used in the *SimulationRunner* class in order to store results from a simulation. It is able to combine the results from multiple iterations (of the *SimulationRunner._run_simulation()* method in the SimulationRunner class) as well as append results for different simulation parameters configurations.

Note: Each result stored in the *SimulationResults* object is in fact an object of the *Result* class. This is required so that multiple *SimulationResults* objects can be merged together, since the logic to merge each individual result is in the the *Result* class.

Examples

- Creating a SimulationResults object and adding a few results to it

```
result1 = Result.create(...)  # See the Result class for details
result2 = Result.create(...)
result3 = Result.create(...)
simresults = SimulationResults()
simresults.add_result(result1)
simresults.add_result(result2)
simresults.add_result(result3)
```

Instead of explicitly create a Result object and add it to the SimulationResults object, we can also create the Result object on-the-fly when adding it to the SimulationResults object by using the *add_new_result()* method.

That is

```
simresults = SimulationResults()
simresults.add_new_result(...)
simresults.add_new_result(...)
simresults.add_new_result(...)
```

- Merging multiple SimulationResults objects

```
# First SimulationResults object
simresults = SimulationResults()
# Create a Result object
result = Result.create('some_name', Result.SUMTYPE, 4)
# and add it to the SimulationResults object.
simresults.add_result(result)

# Second SimulationResults object
simresults2 = SimulationResults()
# We can also create the Result object on-the-fly when adding it
# to the SimulationResults object to save one line.
simresults2.add_new_result('some_name', Result.SUMTYPE, 6)

# We can merge the results in the second SimulationResults object.
# Since the update type of the single result stored is SUMTYPE,
# then the simresults will now have a single Result of SUMTYPE
# type with a value of 10.
simresults.merge_all_results(simresults)
```

See also:

runner.SimulationRunner Base class to implement Monte Carlo simulations.

parameters.SimulationParameters Class to store the simulation parameters.

Result Class to store a single simulation result.

static _from_dict (*d: Dict[str, Any]*) → *pyphysim.simulations.results.SimulationResults*
Convert from a dictionary to a SimulationResults object.

Parameters **d** (*dict*) – The dictionary representing the SimulationResults.

Returns The converted object.

Return type *SimulationResults*

static _load_from_json_file (*filename: str*) → *pyphysim.simulations.results.SimulationResults*

static _load_from_pickle_file (*filename: str*) → *pyphysim.simulations.results.SimulationResults*

_save_to_json (*filename: str*) → None
Save the SimulationResults object to the json file with name *filename*.

Parameters **filename** (*src*) – Name of the file to save the SimulationResults object.

_save_to_pickle (*filename: str*) → None
Save the SimulationResults object to the pickle file with name *filename*.

Parameters **filename** (*src*) – Name of the file to save the SimulationResults object.

_to_dict () → Dict[str, Any]
Convert the SimulationResults object to a dictionary representation.

Returns The dictionary representation of the SimulationResults object.

Return type dict

add_new_result (*name: str*, *update_type: int*, *value: Any*, *total: Any = 0*) → None
Create a new Result object on the fly and add it to the SimulationResults object.

Note: This is Equivalent to the code below,

```
result = Result.create(name, update_type, value, total)
self.add_result(result)
```

which in fact is exactly how this method is implemented.

Parameters

- **name** (*str*) – Name of the Result.
- **update_type** (*int*) – Type of the result (SUMTYPE, RATIOTYPE, MISCTYPE or CHOICETYPE).
- **value** (*any*) – Value of the result.
- **total** (*any | int*) – Total value of the result (used only for the RATIOTYPE and ignored for the other types).

add_result (*result:* pyphysim.simulations.results.Result) → None
Add a result object to the SimulationResults object.

Note: If there is already a result stored with the same name, this will replace it.

Parameters **result** (Result) – The Result object to add to the simulation results.

append_all_results (*other:* pyphysim.simulations.results.SimulationResults) → None
Append all the results of the other SimulationResults object with self.

Parameters **other** (SimulationResults) – Another SimulationResults object

See also:

append_result(), *merge_all_results()*

append_result (*result:* pyphysim.simulations.results.Result) → None
Append a result to the SimulationResults object.

This effectively means that the SimulationResults object will now store a list for the given result name. This allow you, for instance, to store multiple bit error rates with the 'BER' name such that simulation_results_object['BER'] will return a list with the Result objects for each value.

Parameters **result** (Result) – The Result object to append to the simulation results.

Notes

If multiple values for some Result are stored, then only the last value can be updated with *merge_all_results()*.

Raises **ValueError** – If the *result* has a different type from the result previously stored.

See also:

append_all_results(), *merge_all_results()*

get_filename_with_replaced_params (*filename: str*) → str
Perform the string replacements in filename with simulation parameters.

Parameters **filename** (*str*) – Name of the file to save the results. This can have string placements for replacements of simulation parameters. For instance, is *filename* is "somename_{age}.pickle" and the value of an 'age' parameter is '3', then the actual name used to save the file will be "somename_3.pickle"

Returns The name of the file where the results were saved. This will be equivalent to *filename* after string replacements (with the simulation parameters) are done.

Return type string

get_result_names () → List[str]
Get the names of all results stored in the SimulationResults object.

Returns **names** – The names of the results stored in the SimulationResults object.

Return type list[str]

get_result_values_confidence_intervals (*result_name: str*, *P: float = 95.0*, *fixed_params: Optional[Dict[str, Any]] = None*) → List[numpy.ndarray]
Get the values for the results with name *result_name*.

This method is similar to the *get_result_values_list* method, but instead of returning a list with the values it will return a list with the confidence intervals for those values.

Parameters

- **result_name** (*str*) – The name of the desired result.
- **P** (*float*) –
- **fixed_params** (*dict*) – A python dictionary containing the fixed parameters. If *fixed_params* is provided then the returned list will be only a subset of the results that match the fixed values of the parameters in the *fixed_params* argument, where the key is the parameter's name and the value is the fixed value. See the notes in the documentation of *get_result_values_list()* for an example.

Returns **confidence_interval_list** – A list of Numpy (float) arrays. Each element in the list is an array with two elements, corresponding to the lower and upper limits of the confidence interval.8

Return type list[np.ndarray]

See also:

`calc_confidence_interval()`

get_result_values_list(*result_name: str*, *fixed_params: Optional[Dict[str, Any]] = None*) → List[Any]

Get the values for the results with name *result_name*.

Returns a list with the values.

Parameters

- **result_name** (*str*) – The name of the desired result.
- **fixed_params** (*dict*) – A python dictionary containing the fixed parameters. If *fixed_params* is provided then the returned list will be only a subset of the results that match the fixed values of the parameters in the *fixed_params* argument, where the key is the parameter's name and the value is the fixed value. See the notes for an example.

Returns **result_list** – A list with the stored values for the result with name *result_name*

Return type List

Notes

As an example of the usage of the *fixed_params* argument, suppose the results where obtained in a simulation for three parameters: 'first', with value 'A', 'second' with value '[1, 2, 3]' and 'third' with value '[B, C]', where the 'second' and 'third' were set to be unpacked. In that case the returned result list would have a length of 6 (the number of possible combinations of the parameters to be unpacked). If fixed_params is provided with the value of "{'second': 2}" that means that only the subset of results which corresponding to the second parameters having the value of '2' will be provided and the returned list will have a length of 2. If fixed_params is provided with the value of "{'second': '1', 'third': 'C'}" then a single result will be provided instead of a list.

static load_from_file(*filename: str*) → *pyphysim.simulations.results.SimulationResults*

Load the SimulationResults from the file *filename*.

Parameters **filename** (*src*) – Name of the file from where the results will be loaded.

Returns The SimulationResults object loaded from the file *filename*.

Return type *SimulationResults*

merge_all_results (*other:* pyphysim.simulations.results.SimulationResults) → None
Merge all the results of the other SimulationResults object with the results in self.

When there is more then one result with the same name stored in self (for instance two bit error rates -> for different parameters) then only the last one will be merged with the one in *other*. That also means that only one result for that name should be stored in *other*.

Parameters **other** (SimulationResults) – Another SimulationResults object

See also:

append_result(), *append_all_results()*

Notes

This method is used in the SimulationRunner class to combine results of two simulations for the exact same parameters.

property params
Get method for the params property.

save_to_file (*filename: str*) → str
Save the SimulationResults to the file *filename*.

The string in *filename* can have placeholders for string replacements with any parameter value.

Parameters **filename** (*src*) – Name of the file to save the results. This can have string placements for replacements of simulation parameters. For instance, is *filename* is "somename_{age}.pickle" and the value of an 'age' parameter is '3', then the actual name used to save the file will be "somename_3.pickle"

Returns The name of the file where the results were saved. This will be equivalent to *filename* after string replacements (with the simulation parameters) are done.

Return type string

set_parameters (*params:* pyphysim.simulations.parameters.SimulationParameters) → None
Set the parameters of the simulation used to generate the simulation results stored in the SimulationResults object.

Parameters **params** (SimulationParameters) – A SimulationParameters object containing the simulation parameters.

to_dataframe () → pandas.core.frame.DataFrame
Convert the SimulationResults object to a pandas DataFrame.

pyphysim.simulations.results.**combine_simulation_results** (*simresults1:* pyphysim.simulations.results.SimulationResults, *simresults2:* pyphysim.simulations.results.SimulationResults) → *pyphysim.simulations.results.SimulationResults*
Combine two SimulationResults objects with different parameters values.

For this function to work both simulation objects need to have exact the same parameters and only the values of the parameters set to be unpacked can be different.

Parameters

- **simresults1** (SimulationResults) – The first SimulationResults object to be combined.

- **simresults2** (SimulationResults) – The second SimulationResults object to be combined.

Returns The combined SimulationResults object.

Return type *SimulationResults*

Examples

If the first SimulationResults object was obtained for the parameters "p1 = 10" and "p2 = [1, 2, 3]", while the second SimulationResults object was obtained for the parameters "p1 = 10" and "p2 = [2, 4, 6]" and p2 was marked to be unpacked in both of them, then the returned combined SimulationResults object will have parameters "p1 = 10" and "p2 = [1, 2, 3, 4, 6]" with p2 marked to be unpacked.

Note that the results for the values of p2 equal to "2" and "4" exist in both objects and will be merged together.

pyphysim.simulations.runner module

Module containing the simulation runner.

class pyphysim.simulations.runner.**SimulationRunner**(*default_config_file: Optional[str] = None, config_spec: Optional[str] = None, read_command_line_args: bool = True, save_parsed_file: bool = False*)

Bases: object

Base class to run Monte Carlo simulations.

The main idea of the *SimulationRunner* class is that in order to implement a Monte Carlo simulation one would subclass *SimulationRunner* and implement the *_run_simulation()* method (as well as any of the optional methods). The complete procedure is described in the documentation of the simulations module.

Note: If a given run of *_run_simulation* cannot finish and save results for some reason then you should raise a SkipThisOne exception. For instance if your *_run_simulation* implementation inverts a matrix for some reason and in rare cases the matrix you are trying to invert might be singular. You might want too raise a SkipThisOne exception if that occurs to simple "try again".

The code below illustrates the minimum pseudo code to implement a subclass of *SimulationRunner*.

```
class SomeSimulator(SimulationRunner):
def __init__(self):
    super().__init__()
    # Do whatever you need/want

    # Add the simulation parameters to the `params` attribute.
    self.params.add('par1', par1value)
    ...
    self.params.add('parN', parNvalue)
    # Optionally set some parameter(s) to be unpacked
    self.params.set_unpack_parameter('name_of_a_parameter')

def _run_simulation(self, current_parameters):
    # Get the simulation parameters from the current_parameters
    # object. If no parameter was marked to be unpacked, then
```

(continues on next page)

(continued from previous page)

```
    # current_parameters will be equivalent to self.params.
    par1 = current_parameters['par1']
    ...
    parN = current_parameters['parN']

    # Do the simulation of one iteration using the parameters
    # par1,...parN from the current_parameters object.
    ...

    # Save the results of this iteration to a SimulationResults
    # object and return it
    simResults = SimulationResults()
    simResults.add_new_result(...)  # Add one each result you want
    simResults.add_new_result(...)
    return simResults
```

With that, all there is left to run the simulation is to create a SomeSimulator object and call its *simulate()* method.

Parameters

- **default_config_file** (*str*) – Name of the config file. This will be parsed with configobj.
- **config_spec** (*list[str]*) – Configuration specification to validate the config file.
- **read_command_line_args** (*bool*) – If True (default), read and parse command line arguments.
- **save_parsed_file** (*bool*) – If True, the config file will be saved after it is loaded. This is useful to add the default parameters to the config file so that they can be easily changed later. Note that if the config file does not exist at all, then it will be saved regardless of the value of *save_parsed_file*.

See also:

SimulationResults Class to store simulation results.

SimulationParameters Class to store the simulation parameters.

Result Class to store a single simulation result.

static _SimulationRunner__create_default_ipyparallel_view()
Create a default view for parallel computation.

This method is run in the *simulate_in_parallel* method if a "view" is not passed.

_SimulationRunner__run_simulation_and_track_elapsed_time(*current_parameters:* py-physim.simulations.parameters.SimulationP → *py-physim.simulations.results.SimulationResul*
Perform the _run_simulation method and track its execution time. This time will be added as a Result to the returned *SimulationResults* object from _run_simulation.

Parameters **current_parameters** (SimulationParameters) – SimulationParameters object with the parameters for the simulation. The self.params variable is not used directly. It is first unpacked in the simulate function which then calls _run_simulation for each combination of unpacked parameters.

Returns The current simulation results.

Return type *SimulationResults*

Notes

This method is called in the *simulate* and *simulate_in_parallel*.

_keep_going(*current_params:* pyphysim.simulations.parameters.SimulationParameters, *current_sim_results:* pyphysim.simulations.results.SimulationResults, *current_rep:* *int*) → bool

Check if the simulation should continue or stop.

This function may be reimplemented in the derived class if a stop condition besides the number of iterations is desired. The idea is that _run_simulation returns a *SimulationResults* object, which is then passed to _keep_going, which is then in charge of deciding if the simulation should stop or not.

Parameters

- **current_params** (SimulationParameters) – SimulationParameters object with the parameters of the simulation.
- **current_sim_results** (SimulationResults) – SimulationResults object from the last iteration (merged with all the previous results)
- **current_rep** (*int*) – Number of iterations already run.

Returns **result** – True if the simulation should continue or False otherwise.

Return type bool

Notes

This method should be simple (it will be run many times) and SHOULD NOT modify the object.

_on_simulate_current_params_finish(*current_params:* pyphysim.simulations.parameters.SimulationParameters, *current_params_sim_results:* pyphysim.simulations.results.SimulationResults) → None

This method is called once for each simulation parameters combination after all iterations of _run_simulation are performed (for that combination of simulation parameters).

Parameters

- **current_params** (SimulationParameters) – The current combination of simulation parameters.
- **current_params_sim_results** (SimulationResults) – SimulationResults object with the results for the finished simulation with the parameters in current_params.

Notes

Because this method will run inside the _run_simulation method, which is called in a different process when the simulation is performed in parallel, it should only modify member variables that are only used inside _run_simulation.

_on_simulate_current_params_start(*current_params:* pyphysim.simulations.parameters.SimulationParameters) → None

This method is called once for each simulation parameters combination before any iteration of _run_simulation is performed (for that combination of simulation parameters).

Parameters **current_params** (SimulationParameters) – The current combination of simulation parameters.

Notes

Because this method will run inside the _run_simulation method, which is called in a different process when the simulation is performed in parallel, it should only modify member variables that are only used inside _run_simulation. If any other variable is modified these changes WILL NOT be carried back to the original process.

_on_simulate_finish() → None

This method is called only once at the end of the simulate method.

_on_simulate_start() → None

This method is called only once, in the beginning of the the simulate method.

_run_simulation(*current_parameters:* pyphysim.simulations.parameters.SimulationParameters) → *pyphysim.simulations.results.SimulationResults*

Performs one iteration of the simulation.

This function must be implemented in a subclass. It should take the needed parameters from the params class attribute (which was filled in the constructor of the derived class) and return the results as a *SimulationResults* object.

Note that _run_simulation will be called self.rep_max times (or less if an early stop criteria is reached, which requires reimplementing the _keep_going function in the derived class) and the results from multiple repetitions will be merged.

Parameters **current_parameters** (SimulationParameters) – SimulationParameters object with the parameters for the simulation. The self.params variable is not used directly. It is first unpacked in the simulate function which then calls _run_simulation for each combination of unpacked parameters.

Returns **simulation_results** – A SimulationResults object containing the simulation results of the run iteration.

Return type *SimulationResults*

_simulate_common_setup()

Common setup code that must run in the beginning of a simulation.

_simulate_for_current_params_common(*current_params: pyphysim.simulations.parameters.SimulationParameters, update_progress_func: Callable[[int], None] = <function SimulationRunner.<lambda>>*) → Tuple[int, *pyphysim.simulations.results.SimulationResults*, str]

Parameters

- **current_params** (SimulationParameters) – The current parameters
- **update_progress_func** (*(int) -> []*) – The function that can be called to update the current progress.

Returns The value of *current_rep*, the current results as a SimulationResults object, and the name of the file storing partial results.

Return type (int, *SimulationResults*, str)

static _simulate_for_current_params_parallel(*obj:* pyphysim.simulations.runner.SimulationRunner, *current_params:* pyphysim.simulations.parameters.SimulationParameters, *update_progress_func=None*) → Tuple[int, *pyphysim.simulations.results.SimulationResults*, str]

Simulate (parallel) for the current parameters.

Parameters

- **obj** (SimulationRunner) – The same as the self parameter in regular methods. The reason that this method is set to static is to allow it to be pickled.
- **current_params** (SimulationParameters) – The current parameters
- **update_progress_func** (*(int) -> []*) – The function (or an object with __call__ operator) that can be called to update the current progress.

Returns The value of *current_rep*, the current results as a SimulationResults object, and the name of the file storing partial results.

Return type (int, *SimulationResults*, str)

_simulate_for_current_params_serial(*current_params:* pyphysim.simulations.parameters.SimulationParameters) → Tuple[int, *pyphysim.simulations.results.SimulationResults*, str]

Simulate (serial) for the current parameters.

Parameters **current_params** (SimulationParameters) – The current parameters

Returns The value of *current_rep*, the current results as a SimulationResults object, and the name of the file storing partial results.

Return type (int, *SimulationResults*, str)

clear() → None

Clear the SimulationRunner.

This will erase any results from previous simulations as well as other internal variables. The SimulationRunner object will then be as if it was just created, except that the simulation parameters will be kept.

property delete_partial_results_bool

property elapsed_time

Get the simulation elapsed time. Do not set this value.

Returns The elapsed time.

Return type float

property params

property partial_results_folder

property progress_output_type

property progressbar_message

property results

property results_filename

property runned_reps

Get method for the runned_reps property.

Returns The value of the runned_reps property, which stores the number of runned repetitions for each unpacked parameters combination.

Return type list[int]

set_results_filename(*filename: Optional[str] = None*) → None

Set the name of the file where the simulation results will be saved.

This must be done before calling the *simulate* of the *simulate_in_parallel* methods.

The *filename* argument is formatted with the simulation parameters. That is, suppose there are two parameters Nr=2 and Nt=1, then if *filename* is equal to "results_for_{Nr}x{Nt}" the actual name of the file used to store the simulation results will be "results_for_2x1.pickle".

The advantage of setting the name of the results file with *set_results_filename* instead of manually saving the results after the simulation is finished is that partial results will also be saved. Therefore the simulation can be stopped and continued later from these partial results.

Parameters **filename** (*str*) – The name of the file (without extension) where the simulation results will be stored. If not provided the results will not be automatically stored.

simulate(*param_variation_index: Optional[int] = None*) → None

Performs the full Monte Carlo simulation (serially).

Implements the general code for every simulation. Any code specific to a single simulation iteration must be implemented in the _run_simulation method of a subclass of SimulationRunner.

The main idea behind the SimulationRunner class is that the general code in every simulator is implemented in the SimulationRunner class, more specifically in the *simulate* method, while the specific code of a single iteration is implemented in the _run_simulation method in a subclass.

Parameters **param_variation_index** (*int, optional*) – If not provided, the full simulation will be run. If this is provided the simulation will be run only for the parameters variation with index given by *param_variation_index*. In that case, calling the set_results_filename method before the simulate method is required since only the partial results will be saved.

See also:

simulate_in_parallel()

simulate_common_cleaning()

Common code that must run in the end of a simulation

simulate_in_parallel(*view: Optional[Union[ipyparallel.client.view.LoadBalancedView, ipyparallel.client.view.DirectView]] = None*, *wait: bool = True*) → None

Same as the simulate method, but the different parameters configurations are simulated in parallel.

Parameters

- **view** (*LoadBalancedView | DirectView*) – A ´view´ of the IPython engines. The parallel processing will happen by calling the 'map' method of the provided view to simulate in parallel the different configurations of transmission parameters.
- **wait** (*bool*) – If True then the self.wait_parallel_simulation method will be automatically called at the end of simulate_in_parallel. If False, the YOU NEED to manually call self.wait_parallel_simulation at some point after calling simulate_in_parallel.

Notes

There is a limitation regarding the partial simulation results. The partial results files will be saved in the folder where the IPython engines are running, since the "saving part" is performed in an IPython engine. However, the deletion of the files is not performed in by the IPython engines, but by the main python program. Therefore, unless the Ipython engines are running in the same folder where the main python program will be run the partial result files won't be automatically deleted after the simulation is finished.

property update_progress_function_style

wait_parallel_simulation() → None
Wait for the parallel simulation to finish and then update the self.results variable (as well as other internal variables).

exception pyphysim.simulations.runner.**SkipThisOne**(*msg: str*)
Bases: Exception

Exception that can be raised in the *_run_simulation* method to skip the current repetition.

The *simulate* method will not count a run of *_run_simulation* if it throws a *SkipThisOne* exception.

Parameters **msg** (*str*) – The message with more information on why the exception was raised.

pyphysim.simulations.runner.**get_partial_results_filename**(*results_base_filename: str*, *current_params:* pyphysim.simulations.parameters.SimulationParame *partial_results_folder: Optional[str] = None*) → str
Get the name of the file where the partial result will be saved for the unpacked result with index *unpack_index*.

Parameters

- **results_base_filename** (*str*) – Base name for partial result file.
- **current_params** (SimulationParameters) – The current parameters, which must be a "unpacked variation" of another SimulationParameters object.
- **partial_results_folder** (*str, optional*) – The folder where the partial results will be stored.

Returns **partial_results_filename** – The name of the partial results file.

Return type str

pyphysim.simulations.simulationhelpers module

Module implementing helper functions for simulators.

pyphysim.simulations.simulationhelpers.**_add_folder_to_ipython_engines_path**(*client: ipyparallel.client.client.Client*, *folder: str*) → None

Add a folder to sys.path of each ipython engine.

The list of engines is get as a direct view from 'client'.

This will also add the folder to the local python path.

Parameters

- **client** (*Client*) – The client from which we will get a direct view to access the engines.
- **folder** (*str*) – The folder to be added to the python path at each engine.

pyphysim.simulations.simulationhelpers.**_simulate_do_what_i_mean_multiple_runners**(*list_of_runner List[pyphysim folder: Optional[str] = None*) → None

This will either call the *simulate* method or the *simulate_in_parallel* method as appropriated.

If the 'parameters variation index' was specified in the command line, then the *simulate* method will be called with that index. If not, then the *simulate* method will be called without any index or, if there is an ipython cluster running, the *simulate_in_parallel* method will be called.

Parameters

- **list_of_runners** (*list[SimulationRunner]*) – The *_simulate_do_what_i_mean_single_runner* will be called for each object in the list.
- **folder** (*str*) – Folder to be added to the python path. This should be the main pyphysim folder.

pyphysim.simulations.simulationhelpers.**_simulate_do_what_i_mean_single_runner**(*runner:* py-physim.simulation *folder: Optional[str] = None, block: bool = True*) → None

This will either call the *simulate* method or the *simulate_in_parallel* method as appropriated.

If the 'parameters variation index' was specified in the command line, then the *simulate* method will be called with that index. If not, then the *simulate* method will be called without any index or, if there is an ipython cluster running, the *simulate_in_parallel* method will be called.

Parameters

- **runner** (*SimulationRunner*) – The SimulationRunner object for which either the 'simulate' or the 'simulate_in_parallel' method will be called.
- **folder** (*str, optional*) – Folder to be added to the python path. This should be the main pyphysim folder
- **block** (*bool, optional*) – Passed to the simulate_in_parallel method when the simulation is performed in parallel. If this is false, you need to call the method 'wait_parallel_simulation' of the runner object at some point.

pyphysim.simulations.simulationhelpers.**simulate_do_what_i_mean**(*runner_or_list_of_runners: Union[pyphysim.simulations.runner.Sim List[pyphysim.simulations.runner.Simula folder: Optional[str] = None*) → None

This will either call the simulate method or the simulate_in_parallel method as appropriated.

If the 'parameters variation index' was specified in the command line, then the 'simulate' method will be called with that index. If not, then the simulate method will be called without any index or, if there is an ipython cluster running, the simulate_in_parallel method will be called.

Parameters

- **runner_or_list_of_runners** (*SimulationRunner | list[SimulationRunner]*) – The SimulationRunner object for which either the 'simulate' or the 'simulate_in_parallel' method will be called. If this is a list, then we just call this method individually for each member of the list.
- **folder** (*str*) – Folder to be added to the python path. This should be the main pyphysim folder

Module contents

Module containing useful classes to implement Monte Carlo simulations.

The main class for Monte Carlo simulations is the *runner.SimulationRunner* class, but a few other classes are also implemented to handle simulation parameters and simulation results.

More specifically, the *simulations* module implements the classes:

- *runner.SimulationRunner*
- *parameters.SimulationParameters*
- *results.SimulationResults*
- *results.Result*

For a description of how to implement Monte Carlo simulations using the classes defined in the *simulations* module see the section *Implementing Monte Carlo simulations*.

pyphysim.subspace package

Submodules

pyphysim.subspace.metrics module

Implement several metrics for subspaces.

pyphysim.subspace.metrics.**calc_chordal_distance**(*matrix1:* *numpy.ndarray*, *matrix2:* *numpy.ndarray*) → float

Calculates the chordal distance between the two matrices

Parameters

- **matrix1** (*np.ndarray*) – A 2D numpy array.
- **matrix2** (*np.ndarray*) – A 2D numpy array.

Returns **chord_dist** – The chordal distance.

Return type float

Notes

Same as *calc_chordal_distance_2()*, but implemented differently.

See also:

calc_chordal_distance_2(), *calc_chordal_distance_from_principal_angles()*

Examples

```
>>> A = np.arange(1, 9.)
>>> A.shape = (4, 2)
>>> B = np.array([[1.2, 2.1], [2.9, 4.3], [5.2, 6.1], [6.8, 8.1]])
>>> print(round(calc_chordal_distance(A, B), 8))
0.47386786
```

pyphysim.subspace.metrics.**calc_chordal_distance_2**(*matrix1: numpy.ndarray*, *matrix2: numpy.ndarray*) → float

Calculates the chordal distance between the two matrices

Parameters

- **matrix1** (*np.ndarray*) – A 2D numpy array.
- **matrix2** (*np.ndarray*) – A 2D numpy array.

Returns **chord_dist** – The chordal distance.

Return type float

Notes

Same as *calc_chordal_distance()*, but implemented differently.

See also:

calc_chordal_distance(), *calc_chordal_distance_from_principal_angles()*

Examples

```
>>> A = np.arange(1, 9.)
>>> A.shape = (4, 2)
>>> B = np.array([[1.2, 2.1], [2.9, 4.3], [5.2, 6.1], [6.8, 8.1]])
>>> print(round(calc_chordal_distance_2(A, B), 8))
0.47386786
```

pyphysim.subspace.metrics.**calc_chordal_distance_from_principal_angles**(*principalAngles: numpy.ndarray*) → float

Calculates the chordal distance from the principal angles.

It is given by the square root of the sum of the squares of the sin of the principal angles.

Parameters **principalAngles** (*np.ndarray*) – Numpy array with the principal angles. This is a 1D numpy array.

Returns **chord_dist** – The chordal distance.

Return type float

See also:

calc_principal_angles(), *calc_chordal_distance()*, *calc_chordal_distance_2()*

Examples

```
>>> A = np.arange(1, 9.)
>>> A.shape = (4, 2)
>>> B = np.array([[1.2, 2.1], [2.9, 4.3], [5.2, 6.1], [6.8, 8.1]])
>>> princ_angles = calc_principal_angles(A, B)
>>> print(round(calc_chordal_distance_from_principal_angles(princ_angles), 8))
0.47386786
```

pyphysim.subspace.metrics.**calc_principal_angles**(*matrix1: numpy.ndarray*, *matrix2: numpy.ndarray*) → numpy.ndarray

Calculates the principal angles between *matrix1* and *matrix2*.

Parameters

- **matrix1** (*np.ndarray*) – A 2D numpy array.
- **matrix2** (*np.ndarray*) – A 2D numpy array.

Returns The principal angles between *matrix1* and *matrix2*. This is a 1D numpy array.

Return type np.ndarray

See also:

calc_chordal_distance_from_principal_angles()

Examples

```
>>> A = np.array([[1, 2], [3, 4], [5, 6]])
>>> B = np.array([[1, 5], [3, 7], [5, -1]])
>>> print(calc_principal_angles(A, B))
[0.         0.54312217]
```

pyphysim.subspace.projections module

Module related to subspace projection.

class pyphysim.subspace.projections.**Projection**(*A: numpy.ndarray*)

Bases: object

Class to calculate the projection, orthogonal projection and reflection of a given matrix in a Subspace *S* spanned by the columns of a matrix *A*.

The matrix A is provided in the constructor and after that the functions *project*, *oProject* and *reflect* can be called with M as an argument.

Parameters **A** (*np.ndarray*) – The matrix whose columns form a basis for the projected subspace.

Examples

```
>>> A = np.array([[1 + 1j, 2 - 2j], [3 - 2j, 0], [-1 - 1j, 2 - 3j]])
>>> v = np.array([1, 2, 3])
>>> P = Projection(A)
>>> P.project(v)
array([1.69577465+0.87887324j, 1.33802817+0.41408451j,
       2.32957746-0.56901408j])
>>> P.oProject(v)
array([-0.69577465-0.87887324j,  0.66197183-0.41408451j,
        0.67042254+0.56901408j])
>>> P.reflect(v)
array([-2.3915493 -1.75774648j, -0.67605634-0.82816901j,
       -1.65915493+1.13802817j])
```

static calcOrthogonalProjectionMatrix(*A: numpy.ndarray*) → numpy.ndarray

Calculates the projection matrix that projects a vector (or a matrix) into the signal space orthogonal to the signal space spanned by the columns of M.

Parameters **A** (*np.ndarray*) – A matrix whose columns form a basis for the "desired subspace".

Returns The projection matrix that can be used to project a vector or a matrix into the subspace orthogonal to the subspace spanned by the columns of *A*

Return type np.ndarray

See also:

calcProjectionMatrix()

Examples

```
>>> A = np.array([[1, 2], [2, 2], [4, 3]])
>>> # Matrix that projects into the subspace orthogonal to the
>>> # subspace spanned by the columns of A
>>> oQ = calcOrthogonalProjectionMatrix(A)
>>> print(oQ)
[[ 0.12121212 -0.3030303   0.12121212]
 [-0.3030303   0.75757576 -0.3030303 ]
 [ 0.12121212 -0.3030303   0.12121212]]
```

static calcProjectionMatrix(*A: numpy.ndarray*) → numpy.ndarray

Calculates the projection matrix that projects a vector (or a matrix) into the signal space spanned by the columns of *A*.

Parameters **A** (*np.ndarray*) – A matrix whose columns form a basis for the desired subspace.

Returns The projection matrix that can be used to project a vector or a matrix into the subspace spanned by the columns of *A*

Return type np.ndarray

See also:

calcOrthogonalProjectionMatrix()

Examples

```
>>> A = np.array([[1 + 1j, 2 - 2j], [3 - 2j, 0],                          [-
↪1 - 1j, 2 - 3j]])
>>> # Matrix that projects into the subspace spanned by the columns
>>> # of A
>>> Q = calcProjectionMatrix(A)
>>> np.allclose(Q.round(4), np.array(          [[ 0.5239+0.j, 0.0366+0.3296j,
↪ 0.3662+0.0732j],           [ 0.0366-0.3296j, 0.7690+0.j, -0.0789+0.2479j],
↪            [ 0.3662-0.0732j, -0.0789-0.2479j, 0.7070-0.j]]))
True
```

oProject (*M: numpy.ndarray*) → numpy.ndarray
Project the matrix (or vector) M the subspace ORTHOGONAL to the subspace projected with *project*.

Parameters **M** (*np.ndarray*) – The matrix to be projected.

Returns The projection of *M* into the orthogonal subspace.

Return type np.ndarray

project (*M: numpy.ndarray*) → numpy.ndarray
Project the matrix (or vector) M in the desired subspace.

Parameters **M** (*np.ndarray*) – The matrix to be projected.

Returns The projection of *M* into the desired subspace.

Return type np.ndarray

reflect (*M: numpy.ndarray*) → numpy.ndarray
Find the reflection of the matrix in the subspace spanned by the columns of *A*

Parameters **M** (*np.ndarray*) – The matrix to be projected.

Returns The reflection of *M* in the subspace.

Return type np.ndarray

pyphysim.subspace.projections.**calcOrthogonalProjectionMatrix** (*A: numpy.ndarray*) → numpy.ndarray
Calculates the projection matrix that projects a vector (or a matrix) into the signal space orthogonal to the signal space spanned by the columns of M.

Parameters **A** (*np.ndarray*) – A matrix whose columns form a basis for the "desired subspace".

Returns The projection matrix that can be used to project a vector or a matrix into the subspace orthogonal to the subspace spanned by the columns of *A*

Return type np.ndarray

See also:

calcProjectionMatrix()

Examples

```
>>> A = np.array([[1, 2], [2, 2], [4, 3]])
>>> # Matrix that projects into the subspace orthogonal to the
>>> # subspace spanned by the columns of A
>>> oQ = calcOrthogonalProjectionMatrix(A)
>>> print(oQ)
[[ 0.12121212 -0.3030303   0.12121212]
 [-0.3030303   0.75757576 -0.3030303 ]
 [ 0.12121212 -0.3030303   0.12121212]]
```

pyphysim.subspace.projections.**calcProjectionMatrix**(*A: numpy.ndarray*) → numpy.ndarray

Calculates the projection matrix that projects a vector (or a matrix) into the signal space spanned by the columns of *A*.

Parameters **A** (*np.ndarray*) – A matrix whose columns form a basis for the desired subspace.

Returns The projection matrix that can be used to project a vector or a matrix into the subspace spanned by the columns of *A*

Return type np.ndarray

See also:

calcOrthogonalProjectionMatrix()

Examples

```
>>> A = np.array([[1 + 1j, 2 - 2j], [3 - 2j, 0],                          [-1 -␣
↪1j, 2 - 3j]])
>>> # Matrix that projects into the subspace spanned by the columns
>>> # of A
>>> Q = calcProjectionMatrix(A)
>>> np.allclose(Q.round(4), np.array(          [[ 0.5239+0.j, 0.0366+0.3296j, 0.
↪3662+0.0732j],           [ 0.0366-0.3296j, 0.7690+0.j, -0.0789+0.2479j],      ␣
↪     [ 0.3662-0.0732j, -0.0789-0.2479j, 0.7070-0.j]]))
True
```

Module contents

Subspace related stuff.

This includes projections and metrics for subspaces.

pyphysim.util package

Submodules

pyphysim.util.conversion module

Module containing function related to several conversions, such as linear to dB, binary to gray code, as well as the inverse of them.

pyphysim.util.conversion.**EbN0_dB_to_SNR_dB**(*EbN0: NumberOrArray*, *bits_per_symb: int*) → NumberOrArray

Convert an Eb/N0 value (in dB) to the equivalent SNR value (also in dB).

Parameters

- **EbN0** – Eb/N0 value (in dB)
- **bits_per_symb** – Number of bits in a symbol.

Returns SNR value (in dB)

Return type SNR

pyphysim.util.conversion.**SNR_dB_to_EbN0_dB**(*SNR: NumberOrArray*, *bits_per_symb: int*) → NumberOrArray

Convert an SNR value (in dB) to the equivalent Eb/N0 value (also in dB).

Parameters

- **SNR** – SNR value (in dB).
- **bits_per_symb** – Number of bits in a symbol.

Returns Eb/N0 value (in dB)

Return type EbN0

pyphysim.util.conversion.**binary2gray**(*num: IntOrIntArray*) → IntOrIntArray

Convert a number (in decimal format) to the corresponding Gray code (still in decimal format).

Parameters **num** (*int | np.ndarray*) – The number in decimal encoding

Returns **num_gray** – Corresponding gray code (in decimal format) of *num*.

Return type int | np.ndarray

Examples

```
>>> binary2gray(np.arange(0, 8))
array([0, 1, 3, 2, 6, 7, 5, 4])
```

pyphysim.util.conversion.**dB2Linear**(*valueIndB: NumberOrArray*) → NumberOrArray

Convert input from dB to linear scale.

Parameters **valueIndB** (*int | float | np.ndarray*) – Value in dB

Returns **valueInLinear** – Value in Linear scale.

Return type int | float | np.ndarray

Examples

```
>>> dB2Linear(30)
1000.0
```

pyphysim.util.conversion.**dBm2Linear**(*valueIndBm: NumberOrArray*) → NumberOrArray

Convert input from dBm to linear scale.

Parameters **valueIndBm** (*int | float | np.ndarray*) – Value in dBm.

Returns **valueInLinear** – Value in linear scale.

Return type float | np.ndarray

Examples

```
>>> dBm2Linear(60)
1000.0
```

pyphysim.util.conversion.**gray2binary**(*num: IntOrIntArray*) → IntOrIntArray

Convert a number in Gray code (in decimal format) to its original value (in decimal format).

Parameters **num** (*int | np.ndarray*) – The number in gray coding

Returns **num_orig** – The original number (in decimal format) whose Gray code correspondent is *num*.

Return type int | np.ndarray

Examples

```
>>> gray2binary(binary2gray(np.arange(0,10)))
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

pyphysim.util.conversion.**linear2dB**(*valueInLinear: NumberOrArray*) → NumberOrArray

Convert input from linear to dB scale.

Parameters **valueInLinear** (*int | float | np.ndarray*) – Value in Linear scale.

Returns **valueIndB** – Value in dB scale.

Return type int | float | np.ndarray

Examples

```
>>> linear2dB(1000)
30.0
```

pyphysim.util.conversion.**linear2dBm**(*valueInLinear: NumberOrArray*) → NumberOrArray

Convert input from linear to dBm scale.

Parameters **valueInLinear** (*float | np.ndarray*) – Value in Linear scale

Returns **valueIndBm** – Value in dBm.

Return type float | np.ndarray

Examples

```
>>> linear2dBm(1000)
60.0
```

pyphysim.util.conversion.**single_matrix_to_matrix_of_matrices**(*single_matrix: numpy.ndarray*, *nrows: Optional[numpy.ndarray] = None*, *ncols: Optional[numpy.ndarray] = None*) → numpy.ndarray

Converts a single numpy array to a numpy array of numpy arrays.

For instance, a 6x6 numpy array may be converted to a 3x3 numpy array of 2x2 arrays.

Parameters

- **single_matrix** (*np.ndarray*) – The single numpy array. This should be a 1D numpy array or a 2D numpy array.
- **nrows** (*np.ndarray, optional*) – The number of rows of each submatrix (if single_matrix is 2D), or the number of elements in each subarray (if single_matrix is 1D).
- **ncols** (*np.ndarray, optional*) – The number of rows of each submatrix. If *single_matrix* is a 1D array then *ncols* should be None (default)

Returns The converted array (1D or 2D) of arrays (1D or 2D) as a 1D or 2D numpy array of arrays.

Return type np.ndarray

Notes

The parameters *ncols* and *nrows* cannot both be equal to None.

Examples

```
>>> # Case where we have a single array
>>> single_array = np.array([2, 2, 4, 5, 6, 8, 8, 8, 8])
>>> sizes = np.array([2, 3, 4])
>>> m_of_ms = single_matrix_to_matrix_of_matrices(single_array, sizes)
>>> print(m_of_ms.size)
3
>>> print(m_of_ms[0])
[2 2]
>>> print(m_of_ms[1])
[4 5 6]
>>> print(m_of_ms[2])
[8 8 8 8]
>>>
>>> # Case where we have a matrix to break in packs of rows
>>> single_matrix = np.array([[1, 1, 1], [2, 2, 2], [3, 3, 3]])
>>> rows = np.array([1, 2])
>>> multi_M =single_matrix_to_matrix_of_matrices(single_matrix, rows)
>>> print(multi_M[0])
[[1 1 1]]
>>> print(multi_M[1])
[[2 2 2]
 [3 3 3]]
>>> # Case where we have a matrix to break in packs of columns
>>> rows = None
```

(continues on next page)

(continued from previous page)

```
>>> cols = np.array([1, 2])
>>> multi_M=single_matrix_to_matrix_of_matrices(single_matrix,
↪                                    rows, cols)
>>> print(multi_M[0])
[[1]
 [2]
 [3]]
>>> print(multi_M[1])
[[1 1]
 [2 2]
 [3 3]]
>>> # Case where we break into multiple matrices
>>> rows = np.array([2, 1])
>>> cols = np.array([1, 2])
>>> multi_M=single_matrix_to_matrix_of_matrices(single_matrix,
↪                                    rows, cols)
>>> print(multi_M[0, 0])
[[1]
 [2]]
>>> print(multi_M[0, 1])
[[1 1]
 [2 2]]
```

pyphysim.util.misc module

Module containing useful general functions that don't belong to another module.

pyphysim.util.misc.**calc_autocorr**(*x: numpy.ndarray*) → numpy.ndarray

Calculates the (normalized) auto-correlation of an array x starting from lag 0.

Parameters **x** (*np.ndarray*) – A 1D numpy array.

Returns **result** – The normalized auto-correlation of *x*.

Return type np.ndarray

Examples

```
>>> x = np.array([4, 2, 1, 3, 7, 3, 8])
>>> calc_autocorr(x)
array([ 1.   , -0.025,  0.15 , -0.175, -0.25 , -0.2  ,  0.   ])
```

pyphysim.util.misc.**calc_confidence_interval**(*mean: float*, *std: float*, *n: int*, *P: float = 95.0*) → Tuple[float, float]

Calculate the confidence interval that contains the true mean (of a normal random variable) with a certain probability *P*, given the measured *mean*, standard deviation *std* for number of samples *n*.

Only a few values are allowed for the probability *P*, which are: 50%, 60%, 70%, 80%, 90%, 95%, 98%, 99%, 99.5%, 99.8% and 99.9%.

Parameters

- **mean** (*float*) – The measured mean value.
- **std** (*float*) – The measured standard deviation.
- **n** (*int*) – The number of samples used to measure the mean and standard deviation.

- **P** (*float*) – The desired confidence (probability in %) that true value is inside the calculated interval.

Returns A list with two float elements, the interval minimum and maximum values.

Return type float, float

Notes

This function assumes that the estimated random variable is a normal variable.

pyphysim.util.misc.**calc_decorrelation_matrix**(*cov_matrix: numpy.ndarray*) → numpy.ndarray

Calculates the decorrelation matrix that can be applied to a data vector whose covariance matrix is `cov_matrix` so that the new vector covariance matrix is a diagonal matrix.

Parameters **cov_matrix** (*np.ndarray*) – The covariance matrix of the original data that will be decorrelated. This must be a symmetric and positive semi-definite matrix

Returns The decorrelation matrix $_D$. If the original data is a vector it can be decorrelated with $\frac{T}{D}$.

Return type np.ndarray

See also:

calc_whitening_matrix()

pyphysim.util.misc.**calc_shannon_sum_capacity**(*sinrs: Union[numpy.ndarray, float]*) → float

Calculate the sum of the Shannon capacity of the values in *sinrs*

Parameters **sinrs** (*float | np.ndarray*) – SINR values (in linear scale).

Returns **sum_capacity** – Sum capacity.

Return type float

Examples

```
>>> calc_shannon_sum_capacity(11.4)
3.6322682154995127
>>> calc_shannon_sum_capacity(20.3)
4.412781525338476
>>> sinrs_linear = np.array([11.4, 20.3])
>>> print(calc_shannon_sum_capacity(sinrs_linear))
8.045049740837989
```

pyphysim.util.misc.**calc_unorm_autocorr**(*x: numpy.ndarray*) → numpy.ndarray

Calculates the unormalized auto-correlation of an array x starting from lag 0.

Parameters **x** (*np.ndarray*) – A 1D numpy array.

Returns **result** – The unormalized auto-correlation of *x*.

Return type np.ndarray

Examples

```
>>> x = np.array([4, 2, 1, 3, 7, 3, 8])
>>> calc_unorm_autocorr(x)
array([152,  79,  82,  53,  42,  28,  32])
```

pyphysim.util.misc.**calc_whitening_matrix**(*cov_matrix: numpy.ndarray*) → numpy.ndarray

Calculates the whitening matrix that can be applied to a data vector whose covariance matrix is `cov_matrix` so that the new vector covariance matrix is an identity matrix

Parameters **cov_matrix** (*np.ndarray*) – The covariance matrix of the original data that will be decorrelated. This must be a symmetric and positive semi-definite matrix

Returns **whitening_matrix** – The whitening matrix $_W$. If the original data is a vector it can be whitened with $\frac{H}{W}$.

Return type np.ndarray

Notes

The returned *whitening_matrix* matrix will make the covariance of the filtered data an identity matrix. If you only need the the covariance matrix of the filtered data to be a diagonal matrix (not necessarily an identity matrix) what you want to calculate is the "decorrelation matrix". See the *calc_decorrelation_matrix()* function for that.

See also:

calc_decorrelation_matrix()

pyphysim.util.misc.**count_bit_errors**(*first: IntOrIntArray*, *second: IntOrIntArray*, *axis: Optional[Any] = None*) → IntOrIntArray

Compare *first* and *second* and count the number of equivalent bit errors.

The two arguments are assumed to have the index of transmitted and decoded symbols. The count_bit_errors function will compare each element in *first* with the corresponding element in *second*, determine how many bits changed and then return the total number of changes bits. For instance, if we compare the numbers 3 and 0, we see that 2 bits have changed, since 3 corresponds to '11', while 0 corresponds to '00'.

Parameters

- **first** (*int | np.ndarray*) – The decoded symbols.
- **second** (*int | np.ndarray*) – The transmitted symbols.
- **axis** (*int, optional*) – Since first and second can be numpy arrays, when axis is not provided (that is, it is None) then the total number of bit errors of all the elements of the 'difference array' is returned. If axis is provided, then an array of bit errors is returned where the number of bit errors summed along the provided axis is returned.

Returns **bit_errors** – The total number of bit errors.

Return type int | np.ndarray

Examples

```
>>> first = np.array([[2, 3, 3, 0], [1, 3, 1, 2]])
>>> second = np.array([[0, 3, 2, 0], [2, 0, 1, 2]])
>>> # The number of changed bits in each element is equal to
>>> # array([1, 0, 1, 0, 2, 2, 0])
>>> count_bit_errors(first, second)
6
>>> count_bit_errors(first, second, 0)
array([3, 2, 1, 0])
>>> count_bit_errors(first, second, 1)
array([2, 4])
```

pyphysim.util.misc.**equal_dicts**(*a: Dict[Any, Any]*, *b: Dict[Any, Any]*, *ignore_keys: List[Any]*) → bool

Test if two dictionaries are equal ignoring certain keys.

Parameters

- **a** (*dict*) – The first dictionary
- **b** (*dict*) – The second dictionary
- **ignore_keys** (*list*) – A list or tuple with the keys to be ignored.

pyphysim.util.misc.**get_mixed_range_representation**(*array: numpy.ndarray*, *filename_mode: bool = False*) → str

Get the "range representation" of a numpy array. This is similar to get_range_representation, but it no pure range representation is possible it will try to represent as least part of the array as range representations.

Suppose you have the array n = [1, 2, 3, 5, 10, 15, 20, 25, 30, 35, 40, 100]

Except for the 3 initial and the final elements, this array is an arithmetic progression with step equal to 5. Lets keep the 3 initial and the final values and represent the other values as a range representation.

Parameters

- **array** (*np.ndarray*) – The array to be represented as a range expression.
- **filename_mode** (*bool, optional*) – If True, the returned representation will be more suitable to be used as part of a file-name. That is instead of "5:5:40" the string "5_(5)_40" would be returned.

Returns **expr** – A string expression representing *array*.

Return type str

pyphysim.util.misc.**get_principal_component_matrix**(*A: numpy.ndarray*, *num_components: int*) → numpy.ndarray

Returns a matrix without the "principal components" of *A*.

This function returns a new matrix formed by the most significative components of *A*.

Parameters

- **A** (*np.ndarray*) – The original matrix, which is a 2D numpy matrix.
- **num_components** (*int*) – Number of components to be kept.

Returns **out** – The new matrix with the dead dimensions removed.

Return type np.ndarray

Notes

You might want to normalize the returned matrix *out* after calling get_principal_component_matrix to have the same norm as the original matrix *A*.

pyphysim.util.misc.**get_range_representation**(*array: numpy.ndarray*, *filename_mode: bool = False*) → Optional[str]

Get the "range representation" of a numpy array consisting of a arithmetic progression. If no valid range representation exists, return None.

Suppose you have the array n = [5, 10, 15, 20, 25, 30, 35, 40] This array is an arithmetic progression with step equal to 5 and can be represented as "5:5:40", which is exactly what get_range_representation will return for such array.

Parameters

- **array** (*np.ndarray*) – The array to be represented as a range expression.
- **filename_mode** (*bool, optional*) – If True, the returned representation will be more suitable to be used as part of a file-name. That is instead of "5:5:40" the string "5_(5)_40" would be returned.

Returns **expr** – A string expression representing *array*.

Return type str

pyphysim.util.misc.**gmd**(*U: numpy.ndarray*, *S: numpy.ndarray*, *V_H: numpy.ndarray*, *tol: float = 0.0*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

Perform the Geometric Mean Decomposition of a matrix A, whose SVD is given by *[U, S, V_H] = np.linalg.svd(A)*.

The Geometric Mean Decomposition (GMD) is described in paper "Joint Transceiver Design for MIMO Communications Using Geometric Mean Decomposition."

Parameters

- **U** (*np.ndarray*) – First matrix obtained from the SVD decomposition of the original matrix you want to decompose.
- **S** (*np.ndarray*) – Second matrix obtained from the SVD decomposition of the original matrix you want to decompose.
- **V_H** (*np.ndarray*) – Third matrix obtained from the SVD decomposition of the original matrix you want to decompose.
- **tol** (*float*) – The tolerance.

Returns The three matrices *Q*, *R* and *P* such that *A = QRP^H*, *R* is an upper triangular matrix and *Q* and *P* are unitary matrices.

Return type (np.ndarray,np.ndarray,np.ndarray)

pyphysim.util.misc.**int2bits**(*n: int*) → int

Calculates the number of bits needed to represent an integer n.

Parameters **n** (*int*) – The integer number.

Returns **num_bits** – The number of bits required to represent the number *n*.

Return type int

Examples

```
>>> list(map(int2bits, range(0,19)))
[1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5]
```

pyphysim.util.misc.**least_right_singular_vectors**(*A: numpy.ndarray*, *n: int*) → Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

Return the three matrices. The first one is formed by the *n* least significative right singular vectors of *A*, the second one is formed by the remaining right singular vectors of *A* and the third one has the singular values of the singular vectors of the second matrix (the most significative ones).

Parameters

- **A** (*np.ndarray*) – A 2D numpy array.
- **n** (*int*) – An integer between 0 and the number of columns of *A*.

Returns

The tree matrices V0, V1 and S.

The matrix V0 has the right singular vectors corresponding to the *n* least significant singular values.

The matrix V1 has the remaining right singular vectors.

The matrix S has the singular values corresponding to the remaining singular vectors *V1*.

Return type np.ndarray, np.ndarray, np.ndarray

Notes

Because of the sort operation, if you call least_right_singular_vectors(A, ncols_of_A) you will get all the right singular vectors of A with the column order reversed.

Examples

```
>>> A = np.array([1,2,3,6,5,4,2,2,1])
>>> A.shape = (3,3)
>>> (min_Vs, remaining_Vs, S) = least_right_singular_vectors(A,1)
>>> min_Vs
array([[-0.4474985 ],
       [ 0.81116484],
       [-0.3765059 ]])
>>> remaining_Vs
array([[-0.62341491, -0.64116998],
       [ 0.01889071, -0.5845124 ],
       [ 0.78166296, -0.49723869]])
>>> S
array([1.88354706, 9.81370681])
```

pyphysim.util.misc.**leig**(*A: numpy.ndarray*, *n: int*) → Tuple[numpy.ndarray, numpy.ndarray]

Returns a matrix whose columns are the *n* least significant eigenvectors of *A* (eigenvectors corresponding to the *n* dominant eigenvalues).

Parameters

- **A** (*np.ndarray*) – A symmetric matrix (bi-dimensional numpy array)

- **n** (*int*) – Number of desired least significant eigenvectors.

Returns A list with two elements where the first element is a 2D numpy array with the desired eigenvectors, while the second element is a 1D numpy array with the corresponding eigenvalues.

Return type np.ndarray,np.ndarray

Notes

A must be a symmetric matrix so that its eigenvalues are real and positive.

Raises **ValueError** – If *n* is greater than the number of columns of *A*.

Examples

```
>>> A = np.random.randn(3,3) + 1j*np.random.randn(3,3)
>>> V, D = peig(A, 1)
```

pyphysim.util.misc.**level2bits**(*n: int*) → int

Calculates the number of bits needed to represent n different values.

Parameters **n** (*int*) – Number of different levels.

Returns **num_bits** – Number of bits required to represent *n* levels.

Return type int

Examples

```
>>> list(map(level2bits,range(1,20)))
[1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5]
```

pyphysim.util.misc.**peig**(*A: numpy.ndarray*, *n: int*) → Tuple[numpy.ndarray, numpy.ndarray]

Returns a matrix whose columns are the *n* dominant eigenvectors of *A* (eigenvectors corresponding to the *n* dominant eigenvalues).

Parameters

- **A** (*np.ndarray*) – A symmetric matrix (bi-dimensional numpy array).
- **n** (*int*) – Number of desired dominant eigenvectors.

Returns A list with two elements where the first element is a 2D numpy array with the desired eigenvectors, while the second element is a 1D numpy array with the corresponding eigenvalues.

Return type np.ndarray, np.ndarray

Notes

A must be a symmetric matrix so that its eigenvalues are real and positive.

Raises **ValueError** – If *n* is greater than the number of columns of *A*.

Examples

```
>>> A = np.random.randn(3,3) + 1j*np.random.randn(3,3)
>>> V, D = peig(A, 1)
```

pyphysim.util.misc.**pretty_time**(*time_in_seconds: float*) → str

Return the time in a more friendly way.

Parameters **time_in_seconds** (*float*) – Time in seconds.

Returns **time_string** – Pretty time representation as a string.

Return type str

Examples

```
>>> pretty_time(30)
'30.00s'
>>> pretty_time(76)
'1m:16s'
>>> pretty_time(4343)
'1h:12m:23s'
```

pyphysim.util.misc.**qfunc**(*x: float*) → float

Calculates the 'q' function of x.

Parameters **x** (*float*) – The value to apply the Q function.

Returns **result** – Qfunc(x)

Return type float

Examples

```
>>> qfunc(0.0)
0.5
>>> round(qfunc(1.0), 9)
0.158655254
>>> round(qfunc(3.0), 9)
0.001349898
```

pyphysim.util.misc.**randn_c**(**args: int*) → numpy.ndarray

Generates a random circularly complex gaussian matrix.

Parameters ***args** (*any*) – Variable number of arguments (int values) specifying the dimensions of the returned array. This is directly passed to the numpy.random.randn function.

Returns **result** – A random N-dimensional numpy array (complex dtype) where the *N* is equal to the number of parameters passed to *randn_c*.

Return type np.ndarray

Examples

```
>>> a = randn_c(4,3)
>>> a.shape
(4, 3)
>>> a.dtype
dtype('complex128')
```

pyphysim.util.misc.**randn_c_RS**(*RS: numpy.random.mtrand.RandomState*, **args: int*) → numpy.ndarray

Generates a random circularly complex gaussian matrix.

This is essentially the same as the the randn_c function. The only difference is that the randn_c function uses the global RandomState object in numpy, while randn_c_RS use the provided RandomState object. This allow us greater control.

Parameters

- **RS** (*np.random.RandomState*) – The RandomState object used to generate the random values.
- ***args** (*any*) – Variable number of arguments specifying the dimensions of the returned array. This is directly passed to the numpy.random.randn function.

Returns **result** – A random N-dimensional numpy array (complex dtype) where the *N* is equal to the number of parameters passed to *randn_c*.

Return type np.ndarray

pyphysim.util.misc.**replace_dict_values**(*name: str*, *dictionary: Dict[str, str]*, *filename_mode: bool = False*) → str

Perform the replacements in *name* with the value of dictionary[name].

See the usage example below:

```
>>> name = "results_snr_{snr}_param_a_{param_a}"
>>> replacements = {'snr': np.array([0,5,10,15,20]), 'param_a': 'something'}
>>> replace_dict_values(name, replacements)
'results_snr_[0:5:20]_param_a_something'
```

Note that some small changes are performed in the dictionary prior to the replacement. More specifically, modifications such as changing a numpy array to a more compact representation (when possible). This is done by converting the numpy arrays with the get_mixed_range_representation function.

If the string is going to be used as a filename, pass True to *filename_mode* as in the example below

```
>>> replace_dict_values(name, replacements, True)
'results_snr_[0_(5)_20]_param_a_something'
```

Parameters

- **name** (*str*) – The name fo be formatted.
- **dictionary** (*dict*) – The dictionary with the values to be replaced in *name*.
- **filename_mode** (*bool, optional*) – Extra parameter passed to the get_mixed_range_representation function. If True, the returned representation will be more suitable to be used as part of a file-name. That is instead of "5:5:40" the string "5_(5)_40" would be used.

Returns **new_name** – The value of *name* after the replacements in *dictionary*.

Return type str

Examples

```
>>> name = "something {value1} - {value2} something else {value3}"
>>> dictionary = {'value1':'bla bla',                    'value2':np.array([5,␣
↪10, 15, 20, 25, 30]),                    'value3': 76}
>>> replace_dict_values(name, dictionary, True)
'something bla bla - [5_(5)_30] something else 76'
```

pyphysim.util.misc.**update_inv_sum_diag**(*invA: numpy.ndarray*, *diagonal: numpy.ndarray*) → numpy.ndarray

Calculates the inverse of a matrix $(A + D)$, where D is a diagonal matrix, given the inverse of :math`A` and the diagonal of D.

This calculation is performed using the Sherman-Morrison formula, given my

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u},$$

where u and v are vectors.

Parameters

- **invA** (*np.ndarray*) – A 2D numpy array.
- **diagonal** (*np.ndarray*) – A 1D numpy array with the elements in the diagonal of D.

Returns **new_inv** – The inverse of $A + D$.

Return type np.ndarray

pyphysim.util.misc.**xor**(*a: int*, *b: int*) → int

Calculates the xor operation between a and b.

In python this is performed with a^b. However, sage changed the "^" operator. This xor function was created so that it can be used in either sage or in regular python.

Parameters

- **a** (*int*) – First number.
- **b** (*int*) – Second number.

Returns The result of the *xor* operation between *a* and *b*.

Return type int

Examples

```
>>> xor(3,7)
4
>>> xor(15,6)
9
```

pyphysim.util.serialize module

Module containing function related to serialization.

class pyphysim.util.serialize.**JsonSerializable**
Bases: object

Base class for classes you want to be JSON serializable (convert to/from JSON).

You can call the methods *to_json* and *from_json* methods (the later is a staticmethod).

Note that a subclass must implement the *_to_dict* and *_from_dict* methods.

static _from_dict(*d: Dict[str, Any]*) → Any
Convert from a dictionary to an object.

Parameters **d** (*dict*) – The dictionary representing the object.

Returns The converted object.

Return type *Result*

_to_dict() → Dict[str, Any]
Convert the object to a dictionary representation.

Returns The dictionary representation of the object.

Return type dict

classmethod from_dict(*d: Dict[str, Any]*) → Any
Convert from a dictionary to an object.

Parameters **d** (*dict*) – The dictionary representing the Result.

Returns The converted object.

Return type *Result*

classmethod from_json(*data: Any*) → Any
Convert a JSON representation of the object to an actual object.

Parameters **data** (*str*) – The JSON representation of the object.

Returns The actual object

Return type any

to_dict() → Dict[str, Any]
Convert the object to a dictionary representation.

Returns The dictionary representation of the object.

Return type dict

to_json() → Any
Convert the object to JSON.

Returns JSON representation of the object.

Return type str

class pyphysim.util.serialize.**NumpyOrSetEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Bases: json.encoder.JSONEncoder

JSON encoder for numpy arrays.

Pass this class to json.dumps when converting a dictionary to json so that any field which with a numpy array as value will be properly converted.

This encoder will also handle numpy scalars and the native python set types.

When you need to convert the json representation back, use the *json_numpy_or_set_obj_hook* function.

See also:

json_numpy_or_set_obj_hook

default(*obj: Union[numpy.ndarray, numpy.int32, numpy.int64, numpy.float32, numpy.float64, numpy.float128, set]*) → Any

If input object is an ndarray it will be converted into a dict holding data, dtype, _is_numpy_array and shape.

Parameters **obj** (*Serializable*) –

Returns

Return type Serialized Data

pyphysim.util.serialize.**json_numpy_or_set_obj_hook**(*dct: Dict[str, Any]*) → Union[numpy.ndarray, numpy.int32, numpy.int64, numpy.float32, numpy.float64, numpy.float128, set]

Decodes a previously encoded numpy array.

Parameters **dct** (*dict*) – The JSON encoded numpy array.

Returns The decoded numpy array or None if the encoded json data was not an encoded numpy array.

Return type np.ndarray | set | dict, optional

See also:

NumpyOrSetEncoder()

Module contents

Package with several utility modules and classes.

This includes the SimulationRunner class.

Module contents

The Pyphysim library provides many useful classes and functions to perform Monte Carlo simulations of communications systems.

1.2 Implementing Simulators with PyPhysim

Several simulators are already implemented in the "*apps*" package, and can be used as examples of how to implement simulators with the PyPhysim library. The best complete example is probably the "`apps/simulate_psk.py`" file.

In general, the *simulations* module provides a basic framework for implementing Monte Carlo Simulations and implementing a new simulator with PyPhysim starts by subclassing *SimulationRunner* and implementing the *SimulationRunner._run_simulation()* method with the code to simulate a single iteration for that specific simulator.

A few other classes in the simulations module complete the framework by handling simulation parameters and simulation results. The classes in the framwork consist of

- *SimulationRunner*
- *SimulationParameters*
- *SimulationResults*
- *Result*

For a description of how to implement Monte Carlo simulations using the clases defined in the *simulations* module see *Implementing Monte Carlo simulations*.

1.2.1 Getting simulation Parameters from a file

Python has the ConfigParser library (renamed to configparser in Python 3) to parse configuration parameters. You can use it in the *__init__* method of your simulator class (the simulatro main class that inherits from *SimulationRunner*) to read the simulation parameters from a file.

Another good alternative is using the configobj library, which provides an arguable easier to use API then configparser. Another advantage of using configobj is its ability to validate the configuration file.

1.3 Implementing Monte Carlo simulations

A Monte Carlo simulation involves performing the same simulation many times with random samples to calculate the statistical properties of some phenomenon.

The SimulationRunner class makes implementing Monte Carlo simulations easier by implementing much of the necessary code while leaving the specifics of each simulator to be implemented in a subclass.

In the simplest case, in order to implement a simulator one would subclass SimulationRunner, set the simulation parameters in the __init__ method and implement the *_run_simulation()* method with the code to simulate a single iteration for that specific simulator. The simulation can then be performed by calling the *simulate()* method of an object of that derived class. After the simulation is finished, the 'results' parameter of that object will have the simulation results.

The process of implementing a simulator is described in more details in the following.

1.3.1 Simulation Parameters

The simulation parameters can be set in any way as long as they can be accessed in the *_run_simulation()* method. For parameters that won't be changed, a simple way that works is to store these parameters as attributes in the __init__ method.

On the other hand, if you want to run multiple simulations, each one with different values for some of the simulation parameters then store these parameters in the *self.params* attribute and set them to be unpacked (See documentation of the SimulationParameters class for more details). The *simulate()* method will automatically get all possible combinations of parameters and perform a whole Monte Carlo simulation for each of them. The *simulate()* method will pass the 'current_parameters' (a SimulationParameters object) to *_run_simulation()* from where *_run_simulation()* can get the current combination of parameters.

If you want/need to save the simulation parameters for future reference, however, then you should store all the simulation parameters in the *self.params* attribute. This will allow you to call the method *SimulationParameters.save_to_pickled_file()* to save everything into a file. The simulation parameters can be recovered latter from this file by calling the static method *SimulationParameters.load_from_pickled_file()*.

1.3.2 Simulation Results

In the implementation of the *_run_simulation()* method in a subclass of SimulationRunner it is necessary to create an object of the *SimulationResults* class, add each desided result to it (using the *add_result()* method of the *SimulationResults* class) and then return this object at the end of *_run_simulation()*. Note that each result added to this *SimulationResults* object must itself be an object of the *Result* class.

After each run of the *_run_simulation()* method the returned *SimulationResults* object is merged with the *self.results* attribute from where the simulation results can be retreived after the simulation finishes. Note that the way the results from each *_run_simulation()* run are merged together depend on the *update_type* attribute of the *Result* object.

Since you will have the complete simulation results in the self.results object you can easily save them to a file calling its *SimulationResults.save_to_file()* method.

Note: Call the *SimulationResults.set_parameters()* method to set the simulation parameters in the self.results object before calling its save_to_file method. This way you will have information about which simulation parameters were used to generate the results.

1.3.3 Number of iterations the `_run_simulation()` method is performed

The number of times the *_run_simulation()* method is performed for a given parameter combination depend on the *self.rep_max* attribute. It is set by default to '1' and therefore you should set it to the desired value in the __init__ method of the SimulationRunner subclass.

1.3.4 Optional methods

A few methods can be implemented in the SimulationRunner subclass for extra functionalities. The most useful one is probably the *_keep_going()* method, which can speed up the simulation by avoid running unecessary iterations of the *_run_simulation()* method.

Basically, after each iteration of the *_run_simulation()* method the *_keep_going()* method is called. If it returns True then more iterations of *_run_simulation()* will be performed until *_keep_going()* returns False or rep_max iterations are performed. When the *_keep_going()* method is called it receives a *SimulationResults* object with the cumulated results from all iterations so far, which it can then use to decide it the iterations should continue or not.

The other optional methods provide hooks to run code at specific points of the *simulate()* method. They are described briefly below:

- ***SimulationRunner._on_simulate_start()*:** This method is called once at the beginning of the simulate method.
- ***SimulationRunner._on_simulate_finish()*:** This method is called once at the end of the simulate method.
- ***SimulationRunner._on_simulate_current_params_start()*:** This method is called once for each combination of simulation parameters before any iteration of _run_simulation is performed.
- ***SimulationRunner._on_simulate_current_params_finish()*:** This method is called once for each combination of simulation parameters after all iteration of _run_simulation are performed.

At last, for a working example of a simulator, see the `apps/simulate_psk.py` file.

1.3.5 Example of Implementation

See the documentation of the *SimulationRunner* class for a pseudo implementation of a subclass of the *SimulationRunner*.

1.4 Running Simulations in Parallel

If some parameter was marked to be unpacked and instead of calling the *simulate()* method you call the *simulate_in_parallel()* method, then the simulations for the different parameters will be performed in parallel using the parallel capabilities of the IPython interpreter.

In order to call *simulate_in_parallel()* you need to first create a Client (IPython.parallel.Client) and then get a "view" from it. This view is a required argument to call *simulate_in_parallel()*.

The the IPython documentation to understand more.

1.5 Writing Documentation for PyPhysim

The handwritten rst files should be listed in the toctree in the *List of Documentation Articles* section of the `index.rst` file, while the documentation for each package is automatically generated using `sphinx-apidoc` and should be listed in the toctree in the `description.rst` file.

In order to generate the rst files for each package with `sphinx-apidoc` call the command `sphinx-apidoc -o docs pyphysim` from outside the docs folder.

Note that `sphinx-apidoc` will generate a corresponding *.rst* file for each package, that uses the autodoc feature of sphinx to include the docstrings of the python files in the actual documentation. For instance, the `.rst` file of the `comm` module (whose name is `pyphysim.comm.rst`) is shown below.

```
pyphysim.comm package
=====================

Submodules
----------

pyphysim.comm.blockdiagonalization module
-----------------------------------------

.. automodule:: pyphysim.comm.blockdiagonalization
   :members:
   :undoc-members:
   :show-inheritance:

pyphysim.comm.waterfilling module
---------------------------------

.. automodule:: pyphysim.comm.waterfilling
   :members:
   :undoc-members:
   :show-inheritance:


Module contents
---------------

.. automodule:: pyphysim.comm
   :members:
   :undoc-members:
   :show-inheritance:
```

Notice the use of "`.. automodule::`" to get the documentation of the comm package modules from the source code.

Therefore, you only need to run `sphinx-apidoc` when new modules are created in PyPhysim. Everything else will be automatically done by sphinx with the help of the autodoc extension.

1.5.1 Writing math in the documentation

The sphinx documentation generation system is configured to allow the inclusion of math snippets with LaTeX syntax in the documentation. Simply put a directive like `:math:`x+y = \frac{1}{2}`` in the documentation and it will be rendered as $x + y = \frac{1}{2}$ in the final HTML documentation.

In order for this to work you need to download MathJax from *http://www.mathjax.org/download/* and put it in the *docs/Mathjax/* folder.

Also, in order to make writing math easier, a few extra LaTeX macros should be defined. In special, we use bold to indicate matrices. Usually one would need to write `:math:`\mathbf{H}_{jk}`` to write $\mathbf{H}_{jk}$, but the macro "`\mtH`" is used instead for a bold "H" such that we can write the same think with `:math:`\mtH_{jk}`` and this should be configured in Mathjax. Likewise, equivalent macros must also be defined for the other letters as well as `\vtH` and similar to represent vectors (lower case bold letters).

The after downloading Mathjax to the *docs/Mathjax/* edit the *docs/Mathjax/config/local/local.js* file and add the following macros there.

```
// Matrices
TEX.Macro("mtA", "\\mathbf{A}");
TEX.Macro("mtB", "\\mathbf{B}");
TEX.Macro("mtC", "\\mathbf{C}");
TEX.Macro("mtD", "\\mathbf{D}");
TEX.Macro("mtE", "\\mathbf{E}");
TEX.Macro("mtF", "\\mathbf{F}");
TEX.Macro("mtG", "\\mathbf{G}");
TEX.Macro("mtH", "\\mathbf{H}");
TEX.Macro("mtI", "\\mathbf{I}");
TEX.Macro("mtJ", "\\mathbf{J}");
TEX.Macro("mtK", "\\mathbf{K}");
TEX.Macro("mtL", "\\mathbf{L}");
TEX.Macro("mtM", "\\mathbf{M}");
TEX.Macro("mtN", "\\mathbf{N}");
TEX.Macro("mtO", "\\mathbf{P}");
TEX.Macro("mtP", "\\mathbf{P}");
TEX.Macro("mtQ", "\\mathbf{Q}");
TEX.Macro("mtR", "\\mathbf{R}");
TEX.Macro("mtS", "\\mathbf{S}");
TEX.Macro("mtT", "\\mathbf{T}");
TEX.Macro("mtU", "\\mathbf{U}");
TEX.Macro("mtV", "\\mathbf{V}");
TEX.Macro("mtW", "\\mathbf{W}");
TEX.Macro("mtX", "\\mathbf{X}");
TEX.Macro("mtY", "\\mathbf{Y}");
TEX.Macro("mtZ", "\\mathbf{Z}");
// Vectors
TEX.Macro("vtB", "\\mathbf{b}");
TEX.Macro("vtC", "\\mathbf{c}");
TEX.Macro("vtD", "\\mathbf{d}");
TEX.Macro("vtE", "\\mathbf{e}");
TEX.Macro("vtF", "\\mathbf{f}");
TEX.Macro("vtG", "\\mathbf{g}");
TEX.Macro("vtH", "\\mathbf{h}");
TEX.Macro("vtI", "\\mathbf{i}");
TEX.Macro("vtJ", "\\mathbf{j}");
TEX.Macro("vtK", "\\mathbf{k}");
TEX.Macro("vtL", "\\mathbf{l}");
TEX.Macro("vtM", "\\mathbf{m}");
TEX.Macro("vtN", "\\mathbf{n}");
TEX.Macro("vtO", "\\mathbf{p}");
TEX.Macro("vtP", "\\mathbf{p}");
TEX.Macro("vtQ", "\\mathbf{q}");
TEX.Macro("vtR", "\\mathbf{r}");
TEX.Macro("vtS", "\\mathbf{s}");
TEX.Macro("vtT", "\\mathbf{t}");
TEX.Macro("vtU", "\\mathbf{u}");
TEX.Macro("vtV", "\\mathbf{v}");
TEX.Macro("vtW", "\\mathbf{w}");
TEX.Macro("vtX", "\\mathbf{x}");
TEX.Macro("vtY", "\\mathbf{y}");
TEX.Macro("vtZ", "\\mathbf{z}");
```

1.6 Writing Unittests for PyPhysim

The standard `unittest` module is used to implement automated tests for the several packages in the PyPhysim library. The tests are located in the *tests* folder, which should contain a single test file for each package in PyPhysim.

Each test file contains several *test case* classes, as usual in the unittests framework. The first test case class is always a *Doctests test case*, that is, a test case that simple run all the doctests in each module of the package.

After that, a test case class must be implemented for each module in the package, which in turn should test all the classes and functions in that module.

1.6.1 Test Coverage

Ideally unittests should be implemented at the same time the actual code is implemented (or even before that). However, sometimes the tests are implemented the tested code and this may leave code untested.

A good way to make sure that we have a good test coverage in PyPhysim is using the *python-coverage* program. With it we can run all the implemented unittests and get a report of the lines in any module in PyPhysim that was not run by any unittest (thus finding untested code). The *bin* folder contains a script called **run_python_coverage.sh** that will do exactly that.

1.6.2 Code Quality

A good way to ensure code quality, besides implementing unittests, is to employ code analisys tools such as pylint, pep8, pychecker, etc.

For the quality of PyPhysim as a package, see the cheesecake_index tool.

- http://pycheesecake.org/
- http://infinitemonkeycorps.net/docs/pph/

Notes

If you use pylint, this is the .pylintrc file I use (although I don't struggle to fix every pylint warning)

```
[MASTER]

# Specify a configuration file.
#rcfile=

# Python code to execute, usually for sys.path manipulation such as
# pygtk.require().
init-hook='import sys; sys.path.append("./"); sys.path.append("../")'

# Profiled execution.
profile=no

# Add files or directories to the blacklist. They should be base names, not
# paths.
ignore=CVS

# Pickle collected data for later comparisons.
persistent=yes
```

(continues on next page)

(continued from previous page)

```
# List of plugins (as comma separated values of python modules names) to load,
# usually to register additional checkers.
load-plugins=


[MESSAGES CONTROL]

# Enable the message, report, category or checker with the given id(s). You can
# either give multiple identifier separated by comma (,) or put this option
# multiple time.
#enable=

# Disable the message, report, category or checker with the given id(s). You
# can either give multiple identifier separated by comma (,) or put this option
# multiple time (only on the command line, not in the configuration file where
# it should appear only once).
disable=C0301,R0903,C0103,W0621,W0511,W0221,W0141,R0921,W0142,E1101,C0325,C0114
# - The warning W0511 correspond to the TODOs in the code
#
# - The warning W0221 correspond to methods in child classes that
#   re-implement a parent method, but with different arguments
#
# - The warning W0141 corresponds to using built-in functions such as map,
#   filter etc. In fact, the same thing these functions do could be done
#   with list comprehensions, but I like using the map function.

[REPORTS]

# Set the output format. Available formats are text, parseable, colorized, msvs
# (visual studio) and html
output-format=text

# Include message's id in output
include-ids=yes

# Put messages in a separate file for each module / package specified on the
# command line instead of printing them on stdout. Reports (if any) will be
# written in a file name "pylint_global.[txt|html]".
files-output=no

# Tells whether to display a full report or only the messages
reports=yes

# Python expression which should return a note less than 10 (10 is the highest
# note). You have access to the variables errors warning, statement which
# respectively contain the number of errors / warnings messages and the total
# number of statements analyzed. This is used by the global evaluation report
# (RP0004).
evaluation=10.0 - ((float(5 * error + warning + refactor + convention) / statement) *␣
↪10)

# Add a comment according to your evaluation note. This is used by the global
# evaluation report (RP0004).
comment=no


[BASIC]
```

(continues on next page)

(continued from previous page)

```
# Required attributes for module, separated by a comma
required-attributes=

# List of builtins function names that should not be used, separated by a comma
bad-functions=map,filter,apply,input

# Regular expression which should only match correct module names
module-rgx=(([a-z_][a-z0-9_]*)|([A-Z][a-zA-Z0-9]+))$

# Regular expression which should only match correct module level names
const-rgx=(([A-Z_][A-Z0-9_]*)|(__.*__))$

# Regular expression which should only match correct class names
class-rgx=[A-Z_][a-zA-Z0-9]+$

# Regular expression which should only match correct function names
function-rgx=[a-z_][a-z0-9_]{2,30}$

# Regular expression which should only match correct method names
method-rgx=[a-z_][a-z0-9_]{2,30}$

# Regular expression which should only match correct instance attribute names
attr-rgx=[a-z_][a-z0-9_]{2,30}$

# Regular expression which should only match correct argument names
argument-rgx=[a-z_][a-z0-9_]{2,30}$

# Regular expression which should only match correct variable names
variable-rgx=[a-z_][a-z0-9_]{2,30}$

# Regular expression which should only match correct list comprehension /
# generator expression variable names
inlinevar-rgx=[A-Za-z_][A-Za-z0-9_]*$

# Good variable names which should always be accepted, separated by a comma
good-names=i,j,k,ex,Run,_

# Bad variable names which should always be refused, separated by a comma
bad-names=foo,bar,baz,toto,tutu,tata

# Regular expression which should only match functions or classes name which do
# not require a docstring
#no-docstring-rgx=__.*__  # Default


[FORMAT]

# Maximum number of characters on a single line.
max-line-length=85

# Maximum number of lines in a module
max-module-lines=1500

# String used as indentation unit. This is usually " " (4 spaces) or "\t" (1
# tab).
indent-string='    '
```

(continues on next page)

(continued from previous page)

```
[SIMILARITIES]

# Minimum lines number of a similarity.
min-similarity-lines=4

# Ignore comments when computing similarities.
ignore-comments=yes

# Ignore docstrings when computing similarities.
ignore-docstrings=yes


[VARIABLES]

# Tells whether we should check for unused import in __init__ files.
init-import=no

# A regular expression matching the beginning of the name of dummy variables
# (i.e. not used).
dummy-variables-rgx=_|dummy

# List of additional names supposed to be defined in builtins. Remember that
# you should avoid to define new builtins when possible.
additional-builtins=


[MISCELLANEOUS]

# List of note tags to take in consideration, separated by a comma.
notes=FIXME,XXX,TODO


[TYPECHECK]

# Tells whether missing members accessed in mixin class should be ignored. A
# mixin class is detected if its name ends with "mixin" (case insensitive).
ignore-mixin-members=yes

# List of classes names for which member attributes should not be checked
# (useful for classes with attributes dynamically set).
ignored-classes=SQLObject

# When zope mode is activated, add a predefined set of Zope acquired attributes
# to generated-members.
zope=no

# List of members which are set dynamically and missed by pylint inference
# system, and so shouldn't trigger E0201 when accessed. Python regular
# expressions are accepted.
generated-members=REQUEST,acl_users,aq_parent


[DESIGN]

# Maximum number of arguments for function / method
```

(continues on next page)

(continued from previous page)

```
max-args=7

# Argument names that match this expression will be ignored. Default to name
# with leading underscore
ignored-argument-names=_.*

# Maximum number of locals for function / method body
max-locals=15

# Maximum number of return / yield for function / method body
max-returns=6

# Maximum number of branch for function / method body
max-branchs=12

# Maximum number of statements in function / method body
max-statements=50

# Maximum number of parents for a class (see R0901).
max-parents=7

# Maximum number of attributes for a class (see R0902).
max-attributes=10

# Minimum number of public methods for a class (see R0903).
min-public-methods=2

# Maximum number of public methods for a class (see R0904).
max-public-methods=25


[IMPORTS]

# Deprecated modules which should not be used, separated by a comma
deprecated-modules=regsub,string,TERMIOS,Bastion,rexec

# Create a graph of every (i.e. internal and external) dependencies in the
# given file (report RP0402 must not be disabled)
import-graph=

# Create a graph of external dependencies in the given file (report RP0402 must
# not be disabled)
ext-import-graph=

# Create a graph of internal dependencies in the given file (report RP0402 must
# not be disabled)
int-import-graph=


[CLASSES]

# List of interface methods to ignore, separated by a comma. This is used for
# instance to not check methods defines in Zope's Interface base class.
ignore-iface-methods=isImplementedBy,deferred,extends,names,namesAndDescriptions,
↪queryDescriptionFor,getBases,getDescriptionFor,getDoc,getName,getTaggedValue,
↪getTaggedValueTags,isEqualOrExtendedBy,setTaggedValue,isImplementedByInstancesOf,
↪adaptWith,is_implemented_by
```

(continues on next page)

(continued from previous page)

```
# List of method names used to declare (i.e. assign) instance attributes.
defining-attr-methods=__init__,__new__,setUp

# List of valid names for the first argument in a class method.
valid-classmethod-first-arg=cls


[EXCEPTIONS]

# Exceptions that will emit a warning when being caught. Defaults to
# "Exception"
overgeneral-exceptions=Exception
```

1.7 Speeding up PyPhysim

A famous Donald Knuth statements is that "premature optimization is the root of all evil". Before even thinking about optimizing some python code that code should be fully tested with unittests. (see *Writing Unittests for PyPhysim*).

With properly tested code, if the current speed is not enough the first step is to profile the code to identify bottlenecks. The following section describes how to profile code in python.

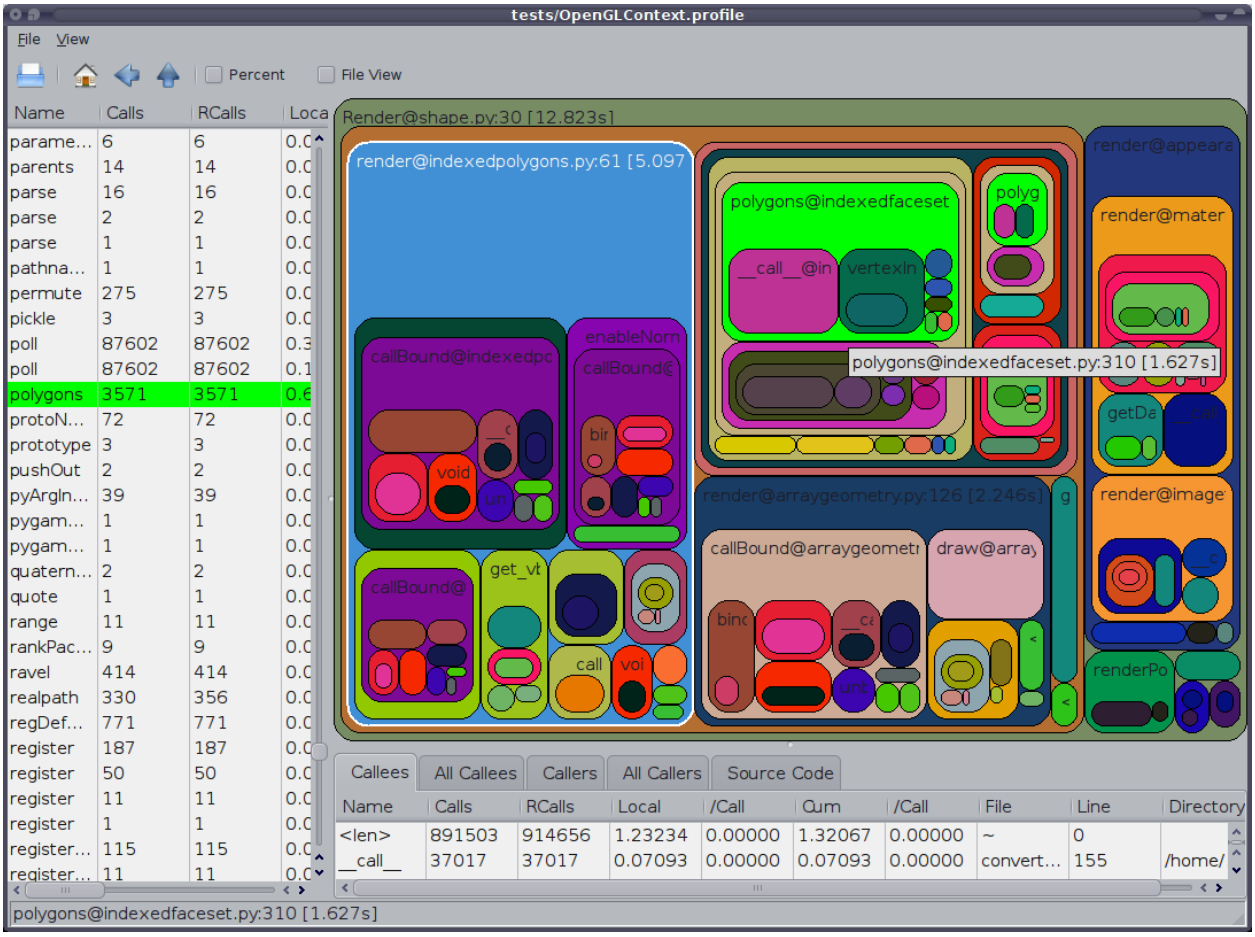
1.7.1 Profiling the Code

There are multiple ways to profile python code and visualize the profile output. An easy way is to use the *%run* command in the ipython interpreter with the *-p* option to run a python script with the python profiler and present the results after the script finishes.

If you prefer to graphically visualize the profile data you can use the runsnakerun program. In order to use it, first you need to run the script using the cProfile module to generate the profile data and then you can visualize the profile data with the runsnakerun program. That is

```
$ python -m cProfile -o <outputfilename> <script-name> <options>
$ runsnake <outputfilename>
```

Here is an image (from http://www.vrplumber.com/programming/runsnakerun/) showing runsnakerun in action.



After identifying which parts in the code need to be optimized, there are multiple ways to achieve faster speeds in python. In PyPhysim we use Cython for that, as described in the following section.

Line Profiler

Another option for profiling is use the line_profiler module. Install the module with

```
$ sudo pip install line_profiler
```

With this the script kernprof.py is also installed. Now, decorate the function you want to profile and run a script that calls the function through kernprof with

```
$ kernprof.py -l -v my_scrypt.py
```

See mode in http://pythonhosted.org/line_profiler/

Memory Profiler

See the blog post below for a way to profile the memory usage of your program. http://www.huyng.com/posts/python-performance-analysis/

1.7.2 Implementing parts of PyPhysim in Cython

Some tutorials about using Cython can be found in - http://docs.cython.org/src/userguide/tutorial.html - http://wiki.cython.org/tutorials/numpy - http://docs.cython.org/src/tutorial/index.html - http://blog.perrygeo.net/2008/04/19/a-quick-cython-introduction/ - http://scipy-lectures.github.com/advanced/advanced_numpy/index.html#exercise-building-an-ufunc-from-scratch - http://wiki.cython.org/PackageHierarchy

The setup.py and setup.cfg files are already properly configured to compile the Cython extensions implement in PyPhysim. Simple call the command

```
$ python setup.py build_ext
```

to compile all the Cython extensions.

In order to describe how the Cython extensions are implemented in PyPhysim lets use the *count_bits* function in the `util.misc` module as an example. We want to make it faster by re-implementing it in Cython.

First the *.count_bits* function was implemented in pure python in `util.misc` and properly tested in the util_package_test.py file along the other functions in `util.misc`.

After that the file *misc_c.pyx* was created, which contains the implementation of *.count_bits* in Cython. Note that the name is equal to the name of the module where the *.count_bits* function originally lives with an added *_c* and the *pyx* extension. The re-implementation in Cython of any function in `util.misc` should be in *misc_c.pyx*.

Then setup.py should be modified to create an extension from the *misc_c.pyx* file (and any other source file it depends on). In the *misc_c.pyx* case this corresponds to adding the code below to setup.py

```
misc_c = Extension(name="misc_c", sources=["util/misc_c.pyx"],
             include_dirs=[numpy.get_include()])
```

and adding "misc_c" to the 'ext_modules' list (an argument of the setup function in the setup.py file).

At last, we add code to the `util.misc` module to use the functions defined in *misc_c.pyx* so that for someone using importing the `util.misc` module it is transparent if the functions are implemented there (in python) or in *misc_c.pyx*. This can be easily done by putting the code below at the end of the *misc.py* file.

```
# xxxxx Load Cython reimplementation of functions here xxxxxxxxxxxxxxxxxxxx
try:
    # If the misc_c.so extension was compiled then any method defined there
    # will replace the corresponding method defined here.
    from c_extensions.misc_c import *
except Exception:
    pass
# xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

The idea is that the user should never import the compiled Cython extension, but only `util.misc`. The code above is enough to replace any functions defined in misc.py by the equivalent function defined in misc_c.pyx whenever the Cython extension is compiled, or use the native python version when the Cython extension is not compiled.

Note: The setup.cfg file is configured so that all the compiled Cython extensions are put in the *c_extensions* folder.

This method has the added benefit that we can run all the unittests on the pure python versions, them compile the Cython extensions and run the unittests again to test the Cython extensions.

Profiling Cython Code

See http://docs.cython.org/src/tutorial/profiling_tutorial.html

You can enable profiling for a Cython source file by putting

```
# cython: profile=True
```

in that source file.

Todo: Verify is this is really necessary when the code is compiled into an extension of only if we had used the pyximport.

Once enabled, your Cython code will behave just like Python code when called from the cProfile module. This means you can just profile your Cython code together with your Python code using the same tools as for Python code alone.

Note: If your profiling is messed up because of the call overhead to some small functions that you rather do not want to see in your profile - either because you plan to inline them anyway or because you are sure that you can't make them any faster - you can use a special decorator to disable profiling for one function only:

```
cimport cython

@cython.profile(False)
def my_often_called_function():
    pass
```

This is important because once my_often_called_function is optimized enough you might want to optimize its calling function and the overhead from profiling my_often_called_function not added to my_often_called_function but to its calling function. Therefore, disabling profiling for my_often_called_function will give you more reliable information when optimizing its calling function.

1.7.3 Other Alternatives to speed-up python code

There are a number of alternatives to speed-up python code.

- you can use the weave module (inline or blitz methods) from scipy to spped up things here. See http://docs.scipy.org/doc/scipy/reference/tutorial/weave.html and http://www.scipy.org/PerformancePython
- You could use Cython
- You could try numexpr http://code.google.com/p/numexpr/
- You could try Numba http://jakevdp.github.com/blog/2012/08/24/numba-vs-cython/
- Use smart numpy broadcast tricks to avoid loops This is fast, but uses more memory. See the source code of the `Modulator.demodulate()` method.
- General tips http://scipy-lectures.github.com/advanced/optimizing/index.html#line-profiler

1.8 Packaging

We use virtualenv and pip.

In the main folder, create an enviroment with

```
$ virtualenv env
```

This will create a *env* folder. Activate the environment with

```
$ source env/bin/activate
```

Now you can install the python dependencies such as numpy, scipy, Cython, IPython, etc.

1.9 Typing Support in PyPhysim

PyPhysim has an increasing support for static typing checking.

Ideally everything in PyPhysim should be type checked without errors by mypy and any new code should ideally have typing information as well.

Note: There are other type checkers that can be used, such as pytype (from Google) or pyre (from Facebook)

1.9.1 Some useful information:

- What is the difference between TypeVar and NewType?
- Covariance, Contravariance, and Invariance — The Ultimate Python Guide
- Python Type Checking (Guide)
- Accepting any derived class: If the number of subclasses is fixed, just create a Union with all of them. If it is not and you truly want "any subclass of Base", then try "Generic[Base]" as the argument type.

```
FadingGenerator = TypeVar('FadingGenerator',
                          bound=FadingSampleGenerator)
```

- Inspecting the type of variables: Use `reveal_type(expr)` to see the type of `expr`. It only works in *mypy* and you get a `NameError` if you try to run code with it in Python.

1.10 References

CHAPTER

TWO

PACKAGES AND MODULES IN PYPHYSIM

2.1 pyphysim

CHAPTER

THREE

TODO LIST

Todo: Verify is this is really necessary when the code is compiled into an extension of only if we had used the pyximport.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/pyphysim/checkouts/latest/docs/hand_written/speedu
line 168.)

BIBLIOGRAPHY

[PetersHeathAltMin2009] Peters, S.W.; Heath, R.W., "Interference alignment via alternating minimization," Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on, pp.2445,2448, 19-24 April 2009

[CadambeDoF2008] V. R. Cadambe and S. A. Jafar, "Interference Alignment and Degrees of Freedom of the K User Interference Channel," IEEE Transactions on Information Theory 54, pp. 3425-3441, Aug. 2008.

[Peters2011] S. W. Peters and R. W. Heath, "Cooperative Algorithms for MIMO Interference Channels," vol. 60, no. 1, pp. 206-218, 2011.

[Bertrand2011] Bertrand, Pierre, "Channel Gain Estimation from Sounding Reference Signal in LTE," Conference: Proceedings of the 73rd IEEE Vehicular Technology Conference.

[Cadambe2008] K. Gomadam, V. R. Cadambe, and S. A. Jafar, "Approaching the Capacity of Wireless Networks through Distributed Interference Alignment," in IEEE GLOBECOM 2008 - 2008 IEEE Global Telecommunications Conference, 2008, pp. 1-6.

[Spencer2004] Q. H. Spencer, A. L. Swindlehurst, and M. Haardt, "Zero-Forcing Methods for Downlink Spatial Multiplexing in Multiuser MIMO Channels," IEEE Transactions on Signal Processing, vol. 52, no. 2, pp. 461-471, Feb. 2004.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

G

H

I

N

O

P

Q

R

S

T

U

V

W

X