BPG Documentation

Release 0.5.3

Pavan Bhargava

Jan 18, 2019

CONTENTS

CHAPTER

ONE

GETTING STARTED

This chapter contains a step-by-step guide to installing BPG and producing your first gds and lsf file. This will utilize the built in example technology information.

1.1 Installing BPG

The first step to setting up BPG is installing the source code and all associated dependencies from github.

1.1.1 Prerequisites

It is highly recommended that you use Anaconda to contain your Python environment setup. This will isolate your system Python installation from your BPG Python installation to minimize unforseen issues. Please install Anaconda with a version >= Python 3.6. This can be found at the Anaconda Website

This guide also assumes that you have a github account with working ssh keys. You will not be able to clone the repositories without it.

1.1.2 Github installation

The Berkeley Analog Generator, which BPG relies on for core infrastructure, requires a very specific file structure and environmental variables.

- To immediately download a fully working file structure, clone the Photonics_Dev repository with `git clone git@github.com:pvnbhargava/Photonics_Dev.git`. This repository should contain several submodules which we will setup and install now:

```
cd Photonics_Dev
git submodule init
git submodule update
```

- Install the included python packages with the pip editable option. This will allow you to pull from git and automatically use any new changes:

```
pip install -e gdspy
pip install -e BPG
pip install -e BAG_framework
```

- You will also need to install a few extra BAG dependencies which are missing from its `setup.py`:

```
conda install shapely
conda install rtree
```

- To test and make sure your installation works properly, launch the bag ipython interpreter and run the provided test suite:

```
sh start_bag.sh
run -i BPG/run_tests.py
```

- You should see something similar to the following output:

```
Photonic_Core_Layout/RingTestSite/tests/test_ring_site.py::test_ring_site
  /Users/cusgadmin/Documents/Photonics_Dev/BPG/BPG/dataprep_gdspy.py:311: RuntimeWarning: [GDSPY] A polygon with more than 199 points w
as created (not officially supported by the GDSII format).
    polygon_out = self.dataprep_cleanup_gdspy(gdspy.PolygonSet(pos_coord_list_list),
  /Users/cusgadmin/Documents/Photonics_Dev/BPG/BPG/dataprep_gdspy.py:235: RuntimeWarning: [GDSPY] A polygon with more than 199 points w
as created (not officially supported by the GDSII format).
    clean_polygon = gdspy.PolygonSet(polygons=clean_coords)
  /Users/cusgadmin/Documents/Photonics_Dev/BPG/BPG/dataprep_gdspy.py:786: RuntimeWarning: [GDSPY] A polygon with more than 199 points w
as created (not officially supported by the GDSII format).
    polygon_out = self.dataprep_cleanup_gdspy(gdspy.PolygonSet(polygon_list),

-- Docs: http://doc.pytest.org/en/latest/warnings.html

Results (68.29s):
      20 passed

In [2]:
```

CHAPTER

TWO

INSTALLATION GUIDE

This chapter contains a more detailed guide on installing BPG for different workspace types.

2.1 Installation Guide

Placeholder text

CHAPTER

THREE

CONFIGURATION FILES

This chapter contains a information on special options to configure the operation of BPG and BAG.

3.1 BPG Configuration

Placeholder text

3.1.1 Anchor1

3.1.2 Anchor2

CHAPTER

FOUR

DATAPREP

This chapter contains an in-depth explanation of how dataprep works, and how to customize the dataprep routine to support photonic layout compilation for your specific PDK

4.1 Dataprep

Placeholder text

CHAPTER

FIVE

BPG

5.1 BPG package

5.1.1 Subpackages

BPG.compiler package

Submodules

BPG.compiler.dataprep_gdspy module

class BPG.compiler.dataprep_gdspy.**Dataprep**(*photonic_tech_info: PhotonicTechInfo*, *grid: RoutingGrid*, *content_list_flat: ContentList*, *is_lsf: bool = False*, *impl_cell=None*)

Bases: object

dataprep() → BPG.content_list.ContentList

Takes the flat content list and performs the specified transformations on the shapes for the purpose of cleaning DRC and prepping tech specific functions.

Notes

1) Take the shapes in the flattened content list and convert them to gdspy format

2) Perform each dataprep operation on the provided layers in order. dataprep_groups is a list where each element contains 2 other lists:

2a) lpp_in defines the layers that the operation will be performed on 2b) lpp_ops defines the operation to be performed 2c) Maps the operation in the spec file to its gdspy implementation and performs it

3) Performs a final over_under_under_over operation

4) Take the dataprepped gdspy shapes and import them into a new post-dataprep content list

dataprep_cleanup_gdspy(*polygon: Union[gdspy.Polygon, gdspy.PolygonSet, None]*, *do_cleanup: bool = True*) → Union[gdspy.Polygon, gdspy.PolygonSet, None]

Clean up a gdspy Polygon/PolygonSet by performing offset with size = 0

First offsets by size 0 with precision higher than the global grid size. Then calls an explicit rounding function to the grid size. This is done because it is unclear how the clipper/gdspy library handles precision

Parameters

- **polygon** (*Union[gdspy.Polygon, gdspy.PolygonSet]*) – The polygon to clean
- **do_cleanup** (*bool*) – True to perform the cleanup. False will return input polygon unchanged

Returns **clean_polygon** – The cleaned up polygon

Return type Union[gdspy.Polygon, gdspy.PolygonSet]

dataprep_coord_to_gdspy(*pos_neg_list_list: Tuple[List[List[Tuple[float, float]]], List[List[Tuple[float, float]]]], manh_grid_size: float, do_manh: bool*) → Union[gdspy.Polygon, gdspy.PolygonSet]

Converts list of polygon coordinate lists into GDSPY polygon objects The expected input list will be a list of all polygons on a given layer

Parameters

- **pos_neg_list_list** (*Tuple[List, List]*) – A tuple containing two lists: the list of positive polygon shapes and the list of negative polygon shapes. Each polygon shape is a list of point tuples
- **manh_grid_size** (*float*) – The Manhattanization grid size
- **do_manh** (*bool*) – True to perform Manhattanization

Returns **polygon_out** – The gdpsy.Polygon formatted polygons

Return type Union[gdspy.Polygon, gdspy.PolygonSet]

dataprep_oversize_gdspy(*polygon: Union[gdspy.Polygon, gdspy.PolygonSet, None], offset: float, do_cleanup: bool = None*) → Union[gdspy.Polygon, gdspy.PolygonSet, None]

Grow a polygon by an offset. Perform cleanup to ensure proper polygon shape.

Parameters

- **polygon** (*Union[gdspy.Polygon, gdspy.PolygonSet, None]*) – The polygon to size, in gdspy representation
- **offset** (*float*) – The amount to grow the polygon
- **do_cleanup** (*bool*) – Optional parameter to force whether point cleanup should occur.

Returns **polygon_oversized** – The oversized polygon

Return type Union[gdspy.Polygon, gdspy.PolygonSet, None]

dataprep_roughsize_gdspy(*polygon: Union[gdspy.Polygon, gdspy.PolygonSet], size_amount: float, do_manh: bool*) → Union[gdspy.Polygon, gdspy.PolygonSet]

Add a new polygon that is rough sized by 'size_amount' from the provided polygon. Rough sizing entails:

- oversize by 2x the global rough grid size
- undersize by 2x the global rough grid size
- oversize by the global rough grid size
- Manhattanize to the global rough grid
- undersize by the fine global fine grid size
- oversize by the fine global fine grid size
- oversize by 'size_amount' less the 2x global grid size already used

Parameters

- **polygon** (*Union[gdspy.Polygon, gdspy.PolygonSet]*) – polygon to be used as the base shape for the rough add, in gdspy representation
- **size_amount** (*float*) – amount to oversize (undersize is not supported, will be set to 0 if negative) the rough added shape
- **do_manh** (*bool*) – True to perform Manhattanization of after the oouuo shape

Returns **polygon_roughsized** – the rough added polygon shapes, in gdspy representation

Return type Union[gdspy.Polygon, gdspy.PolygonSet]

dataprep_undersize_gdspy(*polygon: Union[gdspy.Polygon, gdspy.PolygonSet, None], offset: float, do_cleanup: bool = None*) → Union[gdspy.Polygon, gdspy.PolygonSet, None]

Shrink a polygon by an offset. Perform cleanup to ensure proper polygon shape.

Parameters

- **polygon** (*Union[gdspy.Polygon, gdspy.PolygonSet, None]*) – The polygon to size, in gdspy representation
- **offset** (*float*) – The amount to shrink the polygon
- **do_cleanup** (*bool*) – Optional parameter to force whether point cleanup should occur.

Returns **polygon_undersized** – The undersized polygon

Return type Union[gdspy.Polygon, gdspy.PolygonSet, None]

gdspy_manh(*polygon_gdspy: Union[gdspy.Polygon, gdspy.PolygonSet, None], manh_grid_size: float, do_manh: bool*) → Union[gdspy.Polygon, gdspy.PolygonSet]

Performs Manhattanization on a gdspy representation of a polygon, and returns a gdspy representation of the Manhattanized polygon

Parameters

- **polygon_gdspy** (*Union[gdspy.Polygon, gdspy.PolygonSet, None]*) – The gdspy representation of the polygons to be Manhattanized
- **manh_grid_size** (*float*) – grid size for Manhattanization, edge length after Manhattanization should be larger than it
- **do_manh** (*bool*) – True to perform Manhattanization

Returns **polygon_out** – The Manhattanized polygon, in gdspy representation

Return type Union[gdspy.Polygon, gdspy.PolygonSet]

get_content_on_layer(*layer: Tuple[str, str]*) → BPG.content_list.ContentList

Returns only the content that exists on a given layer

Parameters **layer** (*Tuple[str, str]*) – the layer whose content is desired

Returns **content** – the shape content on the provided layer

Return type *ContentList*

get_manhattanization_size_on_layer(*layer: Union[str, Tuple[str, str]]*) → float

Finds the layer-specific Manhattanization size.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer or LPP being Manhattanized.

Returns **manh_size** – The Manhattanization size for the layer.

Return type float

get_polygon_point_lists_on_layer(*layer: lpp_type*) → Tuple[List[T], List[T]]
Returns a list of all shapes

Parameters **layer** (*Tuple[str, str]*) – the layer purpose pair on which to get all shapes

Returns **positive_polygon_pointlist, negative_polygon_pointlist** – The lists of positive shape and negative shape (holes) polygon boundaries

Return type Tuple[List, List]

static manh_edge_tran(*p1: numpy.ndarray*, *dx: float*, *dy: float*, *nstep: int*, *inc_x_first: bool*, *manh_grid_size: float*, *eps_grid: float = 0.0001*) → numpy.ndarray
Converts pointlist of an edge (ie 2 points), to a pointlist of a Manhattanized edge.

Parameters

- **p1** (*np.ndarray*) – The starting point of the non-Manhattan edge.
- **dx** (*float*) – The x distance to the next point.
- **dy** (*float*) – The y distance to the next point.
- **nstep** (*int*) – The number of steps (each consisting of one horizontal and one vertical segment) that must be added.
- **inc_x_first** (*bool*) – True if the first segment should be horizontal.
- **manh_grid_size** (*float*) – The grid size on which to quantize the steps.
- **eps_grid** (*float*) – The size below which points are considered the same.

Returns **edge_coord_set** – The array of coordinates that define the new Manhattanized edge.

Return type np.ndarray

manh_skill(*poly_coords: Union[List[Tuple[float, float]], numpy.ndarray], manh_grid_size: float, manh_type: str*) → numpy.ndarray
Convert a polygon into a polygon with orthogonal edges (ie, performs Manhattanization)

Parameters

- **poly_coords** (*Union[List[Tuple[float, float]], np.ndarray]*) – list of coordinates that enclose a polygon
- **manh_grid_size** (*float*) – grid size for Manhattanization, edge length after Manhattanization should be larger than it
- **manh_type** (*str*) – 'inc' : the Manhattanized polygon is larger compared to the one on the manh grid 'dec' : the Manhattanized polygon is smaller compared to the one on the manh grid 'non' : additional feature, only map the coords to the manh grid but do no Manhattanization

Returns **poly_coords_cleanup** – The Manhattanized list of coordinates describing the polygon

Return type List[Tuple[float, float]]

static merge_adjacent_duplicate(*coord_set: numpy.ndarray*, *eps_grid: float = 1e-06*) → numpy.ndarray
Merges all points in the passed list of coordinates that are duplicate adjacent points.

Parameters

- **coord_set** (*np.ndarray*) – The input list of coordinates to check for adjacent duplicates.

- **eps_grid** (*float*) – The grid tolerance below which points are considered the same.

Returns **coord_set_merged** – The coordinate list with all adjacent duplicate points removed.

Return type np.ndarray

static not_manh(*coord_list: numpy.ndarray*, *eps_grid: float = 1e-06*) → int

Checks whether the passed coordinate list is Manhattanized

Parameters

- **coord_list** (*List[Tuple[float, float]]*) – The coordinate list to check
- **eps_grid** (*float*) – The grid tolerance below which points are considered the same

Returns **non_manh_edge** – The count of number of edges that are non-Manhattan in this shape

Return type int

poly_operation(*lpp_in: Union[str, Tuple[str, str]], lpp_out: Union[str, Tuple[str, str]], polygon1: Union[gdspy.Polygon, gdspy.PolygonSet, None], polygon2: Union[gdspy.Polygon, gdspy.PolygonSet, None], operation: str, size_amount: Union[float, Tuple[float, float]], do_manh_in_rad: bool = False*) → Union[gdspy.Polygon, gdspy.PolygonSet, None]

Performs a dataprep operation on the input shapes passed by polygon2, and merges (adds/subtracts to/from, replaces, etc) with the shapes currently on the layer passed by polygon1.

The operations implemented in this function must be kept up to date with IMPLEMENTED_DATAPREP_OPERATIONS.

Parameters

- **lpp_in** (*Union[str, Tuple[str, str]]*) – The source layer on which the shapes being added/subtracted are located
- **lpp_out** (*Union[str, Tuple[str, str]]*) – The destination layer on which the shapes are being added to / subtracted from
- **polygon1** (*Union[gdspy.Polygon, gdspy.PolygonSet, None]*) – The shapes currently on the output layer. If operation is manh, polygon1 is the shapes to be Manhattanized
- **polygon2** (*Union[gdspy.Polygon, gdspy.PolygonSet, None]*) – The shapes on the input layer that will be added/subtracted to/from the output layer (ie 'sub' returns (polygon1 - polygon2) on layer lpp_out)
- **operation** (*str*) – The operation to perform: 'manh', 'rad', 'add', 'sub', 'and', 'xor', 'ext', 'ouo'. The implemented functions must match the variable IMPLEMENTED_DATAPREP_OPERATIONS.
- **size_amount** (*Union[float, Tuple[Float, Float]]*) – The amount to over/undersize the shapes to be added/subtracted. For ouo, the 0.5*minWidth related over and under size amount
- **do_manh_in_rad** (*bool*) – True to perform Manhattanization during the 'rad' operation

Returns **polygons_out** – The new polygons present on the output layer

Return type Union[gdspy.Polygon, gdspy.PolygonSet, None]

static polygon_list_by_layer_to_flat_content_list(*poly_list_by_layer: Dict[lpp_type, List[T]], sim_list: List[T], source_list: List[T], monitor_list: List[T], impl_cell: str = 'no_name_cell'*) → ContentList

Converts a LPP-keyed dictionary of polygon pointlists to a flat ContentList format

Parameters

- **poly_list_by_layer** (*Dict[Str, List]*) – A dictionary containing lists all dat-aprepped polygons organized by layername
- **sim_list** (*List*) – The list of simulation boundary content
- **source_list** (*List*) – The list of source object content
- **monitor_list** (*List*) – The list of monitor object content
- **impl_cell** (*str*) – Name of cell in flat gds output

Returns **flat_content_list** – The data in flat content-list-format.

Return type *ContentList*

polyop_gdspy_to_point_list(*polygon_gdspy_in: Union[gdspy.Polygon, gdspy.PolygonSet], fracture: bool = True, do_manh: bool = True, manh_grid_size: Optional[float] = None*) → List[List[Tuple[float, float]]]

Converts the gdspy representation of the polygon into a list of fractured polygon point lists

Parameters

- **polygon_gdspy_in** (*Union[gdspy.Polygon, gdspy.PolygonSet]*) – The gdspy polygons to be converted to lists of coordinates
- **fracture** (*bool*) – True to fracture shapes
- **do_manh** (*bool*) – True to perform Manhattanization
- **manh_grid_size** (*float*) – The Manhattanization grid size

Returns **output_list_of_coord_lists** – A list containing the polygon point lists that compose the input gdspy polygon

Return type List[List[Tuple[float, float]]]

static regex_search_lpps(*regex: Tuple[Pattern[AnyStr], Pattern[AnyStr]], keys: Iterable[Tuple[str, str]]*) → List[Tuple[str, str]]

Returns a list of all keys in the dictionary that match the passed lpp regex. Searches for a match in both the layer and purpose regex.

Parameters

- **regex** (*Tuple[Pattern, Pattern]*) – The lpp regex patterns to match
- **keys** (*Iterable[Tuple[str, str]]*) – The iterable containing the keys of the dictionary.

Returns **matches** – The list of dictionary keys that match the provided regex

Return type List[Tuple[str, str]]

to_polygon_pointlist_from_content_list(*content_list: BPG.content_list.ContentList*) → Tuple[List[T], List[T]]

Convert the provided content list into two lists of polygon pointlists. The first returned list represents the

positive boundaries of polygons. The second returned list represents the 'negative' boundaries of holes in polygons. All shapes in the passed content list are converted, regardless of layer. It is expected that the content list passed to this function only has a single LPP's content

Parameters **content_list** (ContentList) – The content list to be converted to a polygon pointlist

Returns **positive_polygon_pointlist, negative_polygon_pointlist** – The positive shape and negative shape (holes) polygon boundaries

Return type Tuple[List, List]

Notes

No need to loop over content_list, as dataprep only handles a single master at a time No need to handle instance looping, as there are no instances in the flattened content list

BPG.compiler.dataprep_shapely module

BPG.compiler.dataprep_skill module

BPG.compiler.dataprep_skill.**create_global_skill_variables**(*dataprep_procedure_path*, *dataprep_parameters_path*, *output_file_path*)

Parameters

- **dataprep_procedure_path** –
- **dataprep_parameters_path** –

BPG.compiler.dataprep_skill.**setup_bpg_skill**(*output_file_path*, *dataprep_procedure_path*, *dataprep_parameters_path*, *dataprep_skill_function_path*)

BPG.compiler.manh_shapely module

BPG.compiler.manh_shapely.**cleanup_loop**(*coords_list_ori*, *eps_grid=0.0001*)

BPG.compiler.manh_shapely.**coords_apprx_in_line**(*coord1*, *coord2*, *coord3*, *eps_grid=0.0001*)

Tell if three coordinates are in the same line Expected to have three consecutive coordinates as inputs when the function is called

BPG.compiler.manh_shapely.**coords_cleanup**(*coords_list_ori*, *eps_grid=0.0001*, *debug=False*)

clean up coordinates in the list that are redundant or harmful for following Shapely functions

Parameters

- **coords_list_ori** (*list[tuple[float, float]]*) – list of coordinates that enclose a polygon
- **eps_grid** (*float*) – a size smaller than the resolution grid size, if the difference of x/y coordinates of two points is smaller than it, these two points should actually share the same x/y coordinate
- **debug** (*bool*) –

BPG.compiler.manh_shapely.**manh_skill**(*poly_coords*, *manh_grid_size*, *manh_type*)
Convert a polygon into the polygon with orthogonal edges, detailed flavors are the same as it is in the SKILL code

Parameters

- **poly_coords** (*list[tuple[float, float]]*) – list of coordinates that enclose a polygon
- **manh_grid_size** (*float*) – grid size for manhattanization, edge length after manhattanization should be larger than it
- **manh_type** (*str*) – 'inc' : the manhattanized polygon is larger compared to the one on the manh grid 'dec' : the manhattanized polygon is smaller compared to the one on the manh grid 'non' : additional feature, only map the coords to the manh grid but do no manhattanization

BPG.compiler.manh_shapely.**plot_coords**(*ax*, *x*, *y*, *color='#999999'*, *zorder=1*)

BPG.compiler.manh_shapely.**plot_line**(*ax*, *ob*, *color='r'*)

BPG.compiler.manh_shapely.**polyop_manh**(*geom*, *manh_grid_size*, *do_manh*)

BPG.compiler.manh_shapely.**polyop_manh_polygon**(*geom*, *manh_grid_size*, *do_manh*)

BPG.compiler.point_operations module

BPG.compiler.point_operations.**cleanup_delete**(*coords_list_in: numpy.ndarray*, *eps_grid: float = 0.0001*, *cyclic_points: bool = True*, *check_inline: bool = True*) → numpy.ndarray
From the passed coordinate list, returns a numpy array of bools of the same length where each value indicates whether that point should be deleted from the coord_list.

Points that should be removed are either adjacent points that are the same, or points that are in a line.

Parameters

- **coords_list_in** (*np.ndarray*) – The list of x-y coordinates composing a polygon shape
- **eps_grid** – grid resolution below which points are considered to be the same
- **cyclic_points** (*bool*) – True if the coords_list forms a closed polygon. If True, the start/end points might be removed. False if the coords_list is not a closed polygon (ie, a path). If False, the start and end points will never be removed.
- **check_inline** (*bool*) – True [default] to check for and remove center points that are in a line with their two adjacent neighbors. False to skip this check

Returns **delete_array** – Numpy array of bools telling whether to delete the coordinate or not

Return type np.ndarray

BPG.compiler.point_operations.**coords_cleanup**(*coords_list: numpy.ndarray*, *eps_grid: float = 0.0001*, *cyclic_points: bool = True*, *check_inline: bool = True*) → numpy.ndarray
clean up coordinates in the list that are redundant or harmful for following geometry manipulation functions

Points that are cleaned are:

- Adjacent coincident points

- Collinear points (middle points removed)

Parameters

- **coords_list** (*np.ndarray*) – list of coordinates that enclose a polygon
- **eps_grid** (*float*) – a size smaller than the resolution grid size, if the difference of x/y coordinates of two points is smaller than it, these two points should actually share the same x/y coordinate
- **cyclic_points** (*bool*) – True [default] if the coords_list forms a closed polygon. If True, the start/end points might be removed. False if the coords_list is not a closed polygon (ie, a path). If False, the start and end points will never be removed.
- **check_inline** (*bool*) – True [default] to check for and remove center points that are in a line with their two adjacent neighbors. False to skip this check

Returns **coords_set_out** – The cleaned coordinate set

Return type np.ndarray

BPG.compiler.point_operations.**create_polygon_from_path_and_width**(*points_list: numpy.ndarray, width: Union[float, int], eps: float = 0.0001*) → numpy.ndarray

Given a path (a numpy array of 2-D points) and a width (constant along the path), return the set of points forming the polygon.

Checks to see if the radius of curvature is smaller than half the width. If so, the polygon will be self intersecting, so raise an error.

Does not perform any rounding/snapping of points to a grid.

Parameters

- **points_list** (*np.ndarray*) – A numpy array of points (n x 2) representing the center of the path.
- **width** (*Union[float, int]*) – The width of the path
- **eps** (*float*) – The tolerance for determining whether two points are coincident.

Returns **polygon_points** – The polygon formed by the center path and width.

Return type np.ndarray

BPG.compiler.point_operations.**radius_of_curvature**(*pt0*, *pt1*, *pt2*, *eps*) → float

BPG.compiler.poly_simplify module

Module contents

BPG.gds package

Submodules

BPG.gds.core module

class BPG.gds.core.**GDSPlugin**(*grid*, *gds_layermap*, *gds_filepath*, *lib_name*)
Bases: *BPG.abstract_plugin.AbstractPlugin*

export_content_list(*content_lists: List[ContentList]*, *name_append: str = ''*)
Exports the physical design to GDS

Parameters **content_lists** (*List[ContentList]*) – A list of ContentList objects that represent the layout.

Module contents

BPG.lumerical package

Submodules

BPG.lumerical.code_generator module

Module containing classes used to systematically generate clean Lumerical script code

class BPG.lumerical.code_generator.**LumericalCodeGenerator**(*config=None*)
Bases: object

This is the base class that encapsulates the generation of lumerical .lsf code

add_code(*code: str*) → None
Adds provided statement of code to the script file, and formats it accordingly Adds a semicolon and a newline character to each line to match standard LSF syntax

Parameters **code** (*str*) – Single string containing lumerical script

add_formatted_code_block(*code: List[str]*)
Adds a pre-formatted list of lines of code to the script file

Parameters **code** (*List[str]*) – Single string containing lumerical script

add_formatted_line(*code: str*) → None
Adds provided line of code to the script file Does not add a semicolon, but does add a newline character

Parameters **code** (*str*) – Single string containing lumerical script

get_file_header()
Returns a list of strings that form the header of the script file

Returns **header** – Contains comments for the header of the file

Return type List[str]

set(*key: str*, *value*) → None
Conveniently adds a set statement to the LSF file

Parameters

- **key** (*str*) – parameter to be changed with the set statement
- **value** (*any*) – value that the parameter will be assigned

class BPG.lumerical.code_generator.**LumericalDesignGenerator**(*filepath*)
Bases: *BPG.lumerical.code_generator.LumericalCodeGenerator*

export_to_lsf()
Take all code in the database and export it to a lumerical script file

class BPG.lumerical.code_generator.**LumericalMaterialGenerator**(*filepath*)
Bases: *BPG.lumerical.code_generator.LumericalCodeGenerator*

This class enables BPG to create a custom set of materials for use in Lumerical

add_material(*name*) → None
Each time this is called a new material with the provided name is created

Parameters **name** (*str*) – name of the new material being created

add_property(*prop_name*, *prop_value*) → None
Each time this method is called, an lsf line setting the property value to the property name is added

Parameters

- **prop_name** (*str*) – name of the property to be set
- **prop_value** (*Any*) – value of the property to be set

export_to_lsf()

import_material_file(*material_dict*) → None
Takes a dictionary containing other dictionaries and creates an lsf file that defines all of the materials and their properties. Each key in the top level dict is the name of the material, and each value is a dictionary containing the material properties.

Parameters **material_dict** (*dict*) – dict of dicts specifying the materials to be created

import_material_from_dict(*material_name: str*, *prop_dict: dict*) → None
Creates and configures a new material given the properties inside the dictionary.

Parameters

- **material_name** (*str*) – the name of the new material to be created
- **prop_dict** (*dict*) – dict containing all the property info necessary to define the material

class BPG.lumerical.code_generator.**LumericalSweepGenerator**(*filepath*)
Bases: *BPG.lumerical.code_generator.LumericalCodeGenerator*

This class enables the creation of .lsf files for swept variables

add_sweep_point(*script_name*)
Adds a given script name to the be run in the main sweep loop. Scripts are executed in the order in which they are added

Parameters **script_name** (*str*) – Name of script to be executed

create_sweep_loop()

export_to_lsf()
Take all code in the database and export it to a lumerical script file

BPG.lumerical.core module

class BPG.lumerical.core.**LumericalPlugin**(*lsf_export_config*, *lsf_filepath*)
Bases: *BPG.abstract_plugin.AbstractPlugin*

export_content_list(*content_lists: List[ContentList]*)
Exports the physical design into the lumerical LSF format

Parameters **content_lists** (*List[ContentList]*) – A list of flattened content lists that have already been run through lumerical dataprep

BPG.lumerical.objects module

This module contains classes for each type of shape in the content list. These classes define static methods that will convert a content_list representation to a lumerical lsf representation

class BPG.lumerical.objects.**PhotonicAdvancedPolygon**(*resolution*, *layer*, *points*, *negative_points*, *unit_mode=False*)

Bases: bag.layout.objects.Polygon

A layout polygon object.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **layer** (*Union[str, Tuple[str, str]]*) – the layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – the points defining the polygon.
- **unit_mode** (*bool*) – True if the points are given in resolution units.

class BPG.lumerical.objects.**PhotonicBlockage**(*resolution*, *block_type*, *block_layer*, *points*, *unit_mode=False*)

Bases: bag.layout.objects.Blockage

A blockage object. Subclass Polygon for code reuse.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **block_type** (*str*) – the blockage type. Currently supports 'routing' and 'placement'.
- **block_layer** (*str*) – the blockage layer. This value is ignored if blockage type is 'placement'.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – the points defining the blockage.
- **unit_mode** (*bool*) – True if the points are given in resolution units.

classmethod **from_content**(*content*, *resolution*)

class BPG.lumerical.objects.**PhotonicBoundary**(*resolution*, *boundary_type*, *points*, *unit_mode=False*)

Bases: bag.layout.objects.Boundary

A boundary object. Subclasses Polygon for code reuse.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **boundary_type** (*str*) – the boundary type. Currently supports 'PR', 'snap', and 'area'.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – the points defining the blockage.
- **unit_mode** (*bool*) – True if the points are given in resolution units.

classmethod **from_content**(*content*, *resolution*)

class BPG.lumerical.objects.**PhotonicInstance**(*parent_grid: bag.layout.routing.grid.RoutingGrid, lib_name: str, master, loc: Tuple[Union[float, int], Union[float, int]], orient: str, name: str = None, nx: int = 1, ny: int = 1, spx: int = 0, spy: int = 0, unit_mode: bool = False*)

Bases: bag.layout.objects.Instance

A photonic layout instance, with optional arraying parameters. This class adds the ability to read

Parameters

- **parent_grid** (*RoutingGrid*) – the parent RoutingGrid object.
- **lib_name** (*str*) – the layout library name.
- **master** (*TemplateBase*) – the master template of this instance.
- **loc** (*Tuple[Union[float, int], Union[float, int]]*) – the origin of this instance.
- **orient** (*str*) – the orientation of this instance.
- **name** (*Optional[str]*) – name of this instance.
- **nx** (*int*) – number of columns.
- **ny** (*int*) – number of rows.
- **spx** (*Union[float, int]*) – column pitch.
- **spy** (*Union[float, int]*) – row pitch.
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units.

content
A dictionary representation of this instance.

get_bound_box_of(*row=0*, *col=0*)
Returns the bounding box of an instance in this mosaic.

get_photonic_port(*name*, *row=0*, *col=0*)
Returns the photonic port object associated with the provided port name

Parameters

- **name** (*str*) – name of the port to be returned
- **row** (*int*) – row in the array of instances to be accessed
- **col** (*int*) – column in the array of instances to be accessed

Returns **port** – photonic port object associated with the provided name

Return type *PhotonicPort*

get_port_used(*port_name*)

master
The master template of this instance.

move_by(*dx=0*, *dy=0*, *unit_mode=False*)
Move this instance by the given amount.

Parameters

- **dx** (*Union[float, int]*) – the X shift.

- **dy** (*Union[float, int]*) – the Y shift.
- **unit_mode** (*bool*) – True if shifts are given in resolution units

set_port_used(*port_name*)

transform(*loc=(0, 0)*, *orient='R0'*, *unit_mode=False*, *copy=False*)
Transform this figure.

class BPG.lumerical.objects.**PhotonicInstanceInfo**(*res*, *change_orient=True*, ***kwargs*)
Bases: bag.layout.objects.InstanceInfo

A dictionary that represents a layout instance.

content()

copy()
Override copy method of InstanceInfo to return a PhotonicInstanceInfo instead.

master_key

param_list = ['lib', 'cell', 'view', 'name', 'loc', 'orient', 'num_rows', 'num_cols',

class BPG.lumerical.objects.**PhotonicPath**(*resolution*, *layer*, *width*, *points*, *end_style='truncate'*, *join_style='extend'*, *unit_mode=False*)
Bases: bag.layout.objects.Figure

A layout path. Only 45/90 degree turns are allowed.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **layer** (*string or (string, string)*) – the layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **width** (*float*) – width of this path, in layout units.
- **points** (*List[Tuple[float, float]]*) – list of path points.
- **end_style** (*str*) – the path ends style. Currently support 'truncate', 'extend', and 'round'.
- **join_style** (*str*) – the ends style at intermediate points of the path. Currently support 'extend' and 'round'.
- **unit_mode** (*bool*) – True if width and points are given as resolution units instead of layout units.

content
A dictionary representation of this path.

classmethod from_content(*content*, *resolution*)

layer
The rectangle (layer, purpose) pair.

lower

move_by(*dx=0*, *dy=0*, *unit_mode=False*)
Move this path by the given amount.

Parameters

- **dx** (*float*) – the X shift.
- **dy** (*float*) – the Y shift.

- **unit_mode** (*bool*) – True if shifts are given in resolution units.

points

points_unit

classmethod polygon_pointlist_export(*vertices*)

Parameters **vertices** (*List[Tuple[float, float]]*) – The verticies from the content list of this polygon

Returns **output_list** – The positive and negative polygon pointlists describing this polygon

Return type Tuple[List, List]

polygon_points

process_points(*pts*, *width*, *eps=1e-05*, *unit_mode=False*)

Parameters

- **pts** –
- **width** –
- **eps** –
- **unit_mode** –

transform(*loc=(0, 0)*, *orient='R0'*, *unit_mode=False*, *copy=False*)
Transform this figure.

upper

valid
Returns True if this instance is valid.

width

width_unit

class BPG.lumerical.objects.**PhotonicPathCollection**(*resolution*, *paths*)
Bases: bag.layout.objects.PathCollection

A layout figure that consists of one or more paths.

This class make it easy to draw bus/trasmission line objects.

Parameters

- **resolution** (*float*) – layout unit resolution.
- **paths** (*List[Path]*) – paths in this collection.

class BPG.lumerical.objects.**PhotonicPinInfo**(*res*, ***kwargs*)
Bases: bag.layout.objects.PinInfo

A dictionary that represents a layout pin.

classmethod from_content(*content*, *resolution*)

param_list = ['net_name', 'pin_name', 'label', 'layer', 'bbox', 'make_rect']

transform(*loc*, *orient*, *unit_mode*, *copy*)

class BPG.lumerical.objects.**PhotonicPolygon**(*resolution*, *layer*, *points*, *unit_mode=False*)
Bases: bag.layout.objects.Polygon

A layout polygon object.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **layer** (*Union[str, Tuple[str, str]]*) – the layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – the points defining the polygon.
- **unit_mode** (*bool*) – True if the points are given in resolution units.

classmethod from_content(*content*, *resolution*)

classmethod lsf_export(*vertices*, *layer_prop*) → List[str]

Describes the current polygon shape in terms of lsf parameters for lumerical use

Parameters

- **vertices** (*List[Tuple[float, float]]*) – ordered list of x,y coordinates representing the points of the polygon
- **layer_prop** (*dict*) – dictionary containing material properties for the desired layer

Returns **lsf_code** – list of str containing the lsf code required to create specified rectangles

Return type List[str]

classmethod polygon_pointlist_export(*vertices*)

Parameters **vertices** (*List[Tuple[float, float]]*) – The verticies from the content list of this polygon

Returns **output_list** – The positive and negative polygon pointlists describing this polygon

Return type Tuple[List, List]

class BPG.lumerical.objects.**PhotonicRect**(*layer*, *bbox*, *nx=1*, *ny=1*, *spx=0*, *spy=0*, *unit_mode=False*)

Bases: bag.layout.objects.Rect

A layout rectangle, with optional arraying parameters.

Parameters

- **layer** (*string or (string, string)*) – the layer name, or a tuple of layer name and purpose name. If pupose name not given, defaults to 'drawing'.
- **bbox** (*bag.layout.util.BBox or bag.layout.util.BBoxArray*) – the base bounding box. If this is a BBoxArray, the BBoxArray's arraying parameters are used.
- **nx** (*int*) – number of columns.
- **ny** (*int*) – number of rows.
- **spx** (*float*) – column pitch.
- **spy** (*float*) – row pitch.
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units.

classmethod from_content(*content*, *resolution*)

classmethod lsf_export(*bbox*, *layer_prop*, *nx=1*, *ny=1*, *spx=0.0*, *spy=0.0*) → List[str]

Describes the current rectangle shape in terms of lsf parameters for lumerical use. Note that Lumerical uses meters as the base unit, and all input coords are assumed to be in microns. This method inherently resizes

Parameters

- **bbox** (*[[float, float], [float, float]]*) – lower left and upper right corner xy coordinates
- **layer_prop** (*dict*) – dictionary containing material properties for the desired layer
- **nx** (*int*) – number of arrayed rectangles in the x-direction
- **ny** (*int*) – number of arrayed rectangles in the y-direction
- **spx** (*float*) – space between arrayed rectangles in the x-direction
- **spy** (*float*) – space between arrayed rectangles in the y-direction

Returns **lsf_code** – list of str containing the lsf code required to create specified rectangles

Return type List[str]

classmethod polygon_pointlist_export(*bbox*, *nx=1*, *ny=1*, *spx=0.0*, *spy=0.0*)

Convert the PhotonicRect geometry to a list of polygon pointlists.

Parameters

- **bbox** (*[[float, float], [float, float]]*) – lower left and upper right corner xy coordinates
- **nx** (*int*) – number of arrayed rectangles in the x-direction
- **ny** (*int*) – number of arrayed rectangles in the y-direction
- **spx** (*float*) – space between arrayed rectangles in the x-direction
- **spy** (*float*) – space between arrayed rectangles in the y-direction

Returns **output_list** – The positive and negative polygon pointlists describing the photonicRect

Return type Tuple[List, List]

class BPG.lumerical.objects.**PhotonicRound**(*layer*, *resolution*, *center*, *rout*, *rin=0*, *theta0=0*, *theta1=360*, *nx=1*, *ny=1*, *spx=0*, *spy=0*, *unit_mode=False*)

Bases: bag.layout.objects.Arrayable

A layout round object, with optional arraying parameters.

Parameters

- **layer** (*string or (string, string)*) – the layer name, or a tuple of layer name and purpose name. If pupose name not given, defaults to 'drawing'.
- **rout** –
- **rin** –
- **theta0** –
- **theta1** –
- **nx** (*int*) – number of columns.
- **ny** (*int*) – number of rows.
- **spx** (*float*) – column pitch.
- **spy** (*float*) – row pitch.
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units.

center
The center in layout units

center_unit
The center in resolution units

content
A dictionary representation of this rectangle.

classmethod from_content(*content*, *resolution*)

layer
The rectangle (layer, purpose) pair.

classmethod lsf_export(*rout*, *rin*, *theta0*, *theta1*, *layer_prop*, *center*, *nx=1*, *ny=1*, *spx=0.0*, *spy=0.0*)

Parameters

- **rout** –
- **rin** –
- **theta0** –
- **theta1** –
- **layer_prop** –
- **center** –
- **nx** –
- **ny** –
- **spx** –
- **spy** –

move_by(*dx=0*, *dy=0*, *unit_mode=False*)
Moves the round object

static num_of_sparse_point_round(*radius*, *res_grid_size*)

classmethod polygon_pointlist_export(*rout*, *rin*, *theta0*, *theta1*, *center*, *nx=1*, *ny=1*, *spx=0.0*, *spy=0.0*, *resolution=0.001*)

rin
The inner radius in layout units

rin_unit
The inner radius in resolution units

rout
The outer radius in layout units

rout_unit
The outer radius in resolution units

theta0
The starting angle, in degrees

theta1
The ending angle, in degrees

transform(*loc=(0, 0)*, *orient='R0'*, *unit_mode=False*, *copy=False*)
Transform this figure.

class BPG.lumerical.objects.**PhotonicTLineBus**(*resolution*, *layer*, *points*, *widths*, *spaces*, *end_style='truncate'*, *unit_mode=False*)

Bases: bag.layout.objects.TLineBus

A transmission line bus drawn using Path.

assumes only 45 degree turns are used, and begin and end line segments are straight.

Parameters

- **resolution** (*float*) – layout unit resolution.
- **layer** (*Union[str, Tuple[str, str]]*) – the bus layer.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – list of center points of the bus.
- **widths** (*List[Union[float, int]]*) – list of wire widths. 0 index is left/bottom most wire.
- **spaces** (*List[Union[float, int]]*) – list of wire spacings.
- **end_style** (*str*) – the path ends style. Currently support 'truncate', 'extend', and 'round'.
- **unit_mode** (*bool*) – True if width and points are given as resolution units instead of layout units.

class BPG.lumerical.objects.**PhotonicVia**(*tech*, *bbox*, *bot_layer*, *top_layer*, *bot_dir*, *nx=1*, *ny=1*, *spx=0*, *spy=0*, *extend=True*, *top_dir=None*, *unit_mode=False*)

Bases: bag.layout.objects.Via

A layout via, with optional arraying parameters.

Parameters

- **tech** (*bag.layout.core.TechInfo*) – the technology class used to calculate via information.
- **bbox** (*bag.layout.util.BBox or bag.layout.util.BBoxArray*) – the via bounding box, not including extensions. If this is a BBoxArray, the BBoxArray's arraying parameters are used.
- **bot_layer** (*str or (str, str)*) – the bottom layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **top_layer** (*str or (str, str)*) – the top layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **bot_dir** (*str*) – the bottom layer extension direction. Either 'x' or 'y'.
- **nx** (*int*) – arraying parameter. Number of columns.
- **ny** (*int*) – arraying parameter. Mumber of rows.
- **spx** (*float*) – arraying parameter. Column pitch.
- **spy** (*float*) – arraying parameter. Row pitch.
- **extend** (*bool*) – True if via extension can be drawn outside of bounding box.
- **top_dir** (*Optional[str]*) – top layer extension direction. Can force to extend in same direction as bottom.
- **unit_mode** (*bool*) – True if array pitches are given in resolution units.

classmethod from_content(*content*)

class BPG.lumerical.objects.**PhotonicViaInfo**(*res*, ***kwargs*)
Bases: bag.layout.objects.ViaInfo

A dictionary that represents a layout via.

param_list = ['id', 'loc', 'orient', 'num_rows', 'num_cols', 'sp_rows', 'sp_cols', 'en

BPG.lumerical.simulation module

Module containing simulation objects that can be added in lumerical

class BPG.lumerical.simulation.**FDESolver**
Bases: *BPG.lumerical.simulation.LumericalSimObj*

Lumerical Simulation Object for Finite-Difference Eigenmode Solver

align_to_port(*port*, *offset=(0, 0)*, *align_orient=True*)
Moves the center of the simulation object to align to the provided photonic port. Overrides the superclass method to support setting the orientation to match port

Parameters

- **port** (PhotonicPort) – Photonic port for the simulation object to be aligned to
- **offset** (*Tuple*) – (x, y) offset relative to the port location
- **align_orient** (*bool*) – True to set the orientation to match the port orientation, False to ignore port orientation

lsf_export()
Returns a list of Lumerical code describing the creation of a FDESolver object

Returns **lsf_code** – list of Lumerical code to create the FDESolver object

Return type List[str]

mesh_size

orientation

class BPG.lumerical.simulation.**FDTDSolver**
Bases: *BPG.lumerical.simulation.LumericalSimObj*

Lumerical Simulation Object for Finite-Difference Time Domain Solver

geometry
This property returns an object that describes the sizing and placement of the lumerical object. This enables users to easily access this object for more complex geometric manipulation.

classmethod get_default_property_values() → dict
Returns a dictionary containing available properties and their corresponding default values This method is called upon initialization of a LumericalSimObj to populate the property dictionary

lsf_export() → List[str]
Returns a list of Lumerical code describing the creation of a FDESolver object

Returns **lsf_code** – list of Lumerical code to create the FDESolver object

Return type List[str]

class BPG.lumerical.simulation.**LumericalCodeObj**
Bases: *BPG.lumerical.simulation.LumericalSimObj*

Class that enables the easy addition of arbitrary lumerical code

geometry
This property returns an object that describes the sizing and placement of the lumerical object. This enables users to easily access this object for more complex geometric manipulation.

classmethod get_default_property_values()
This class is designed to enable arbitrary code execution so there are no default properties

lsf_export()
Simply return code that the user has explicitly added

class BPG.lumerical.simulation.**LumericalSimObj**
Bases: *BPG.lumerical.code_generator.LumericalCodeGenerator*

Abstract Base Class for all simulation/monitor objects in Lumerical

Notes

- All simulation objects are treated as dictionaries that store their properties and values. These can be modified

directly by users so that the code looks similar to standard lumerical script, or specialized methods can be used to simplify common operations like convergence tests.

- Because only specific properties are available for each type of simulation object, the built-in dict's keys are

set to be immutable. The available properties and their default values are defined by the abstract method get_default_property_values().

- Since geometry manipulation is such a common operation for simulation classes, an abstract property

self.geometry is provided. This property will contain an object that enables quick and easy functions like alignment to ports.

- The final abstract method is lsf_export. The method is called at the end of the build process to convert all of

the values in the property dictionary into their corresponding lumerical code.

content
Calls the lsf_export method to convert the object properties to lumerical code, then formats it into a content dictionary

geometry
This property returns an object that describes the sizing and placement of the lumerical object. This enables users to easily access this object for more complex geometric manipulation.

classmethod get_default_property_values() → dict
Returns a dictionary containing available properties and their corresponding default values This method is called upon initialization of a LumericalSimObj to populate the property dictionary

lsf_export() → List[str]
Returns a list of Lumerical code strings describing the creation of the simulation object. All lumerical objects must implement this method in order to convert their internal object representation into valid code

valid
Add syntax check here to confirm all properties are specified

Type For now all objects are valid. TODO

class BPG.lumerical.simulation.**ModeSource**
Bases: *BPG.lumerical.simulation.LumericalSimObj*

Lumerical Simulation Object that controls and places a power monitor

geometry
This property returns an object that describes the sizing and placement of the lumerical object. This enables users to easily access this object for more complex geometric manipulation.

classmethod get_default_property_values() → dict
Returns a dictionary containing available properties and their corresponding default values This method is called upon initialization of a LumericalSimObj to populate the property dictionary

lsf_export()
Returns a list of Lumerical code strings describing the creation of the simulation object. All lumerical objects must implement this method in order to convert their internal object representation into valid code

class BPG.lumerical.simulation.**PowerMonitor**
Bases: *BPG.lumerical.simulation.LumericalSimObj*

Lumerical Simulation Object that describes a power monitor

geometry
This property returns an object that describes the sizing and placement of the lumerical object. This enables users to easily access this object for more complex geometric manipulation.

classmethod get_default_property_values() → dict
Returns a dictionary containing available properties and their corresponding default values This method is called upon initialization of a LumericalSimObj to populate the property dictionary

lsf_export() → List[str]
Returns a list of Lumerical code describing the creation of a PowerMonitor object

Returns **lsf_code** – list of Lumerical code to create the FDESolver object

Return type List[str]

BPG.lumerical.testbench module

class BPG.lumerical.testbench.**LumericalTB**(*temp_db*, *lib_name*, *params*, *used_names*, ***kwargs*)
Bases: *BPG.template.PhotonicTemplateBase*

Abstract base class for all lumerical testbench generators. This class is structured similarly to layout generators, allowing users to add lumerical objects to a running database. These objects can then be manipulated by the user to setup the desired properties as needed.

add_EME_port()

add_EME_solver()

add_FDE_solver() → BPG.lumerical.simulation.FDESolver
Create a blank FDE solver, add it to the db, and return it to the user for manipulation

add_FDTD_port()

add_FDTD_solver() → BPG.lumerical.simulation.FDTDSolver
Create a blank FDTD solver, add it to the db, and return it to the user for manipulation

add_code_obj()
Create an object that simplifies the process of adding arbitrary lumerical code

add_effective_index_monitor()

add_eme_profile()

add_freq_domain_monitor()

add_gaussian_source()

add_index_monitor()

add_mode_expansion_monitor()

add_mode_source()

add_movie_monitor()

add_point_source()

add_time_domain_monitor()

add_total_field_source()

add_var_FDTD_solver()

construct_tb()
Override this method to specify the procedure for generating the testbench and simulation. YOU MUST IMPLEMENT THIS METHOD TO CREATE A TESTBENCH!

create_dut()
Create and place the provided layout class and parameters at the origin

draw_layout()
This method is used internally to assemble the instance and the TB sources. DO NOT CALL THIS

classmethod get_params_info()
Returns a dictionary from parameter names to descriptions.

Returns **param_info** – dictionary from parameter names to descriptions.

Return type Optional[Dict[str, str]]

plane_wave_source()

Module contents

BPG.oa package

Submodules

BPG.oa.core module

class BPG.oa.core.**OAPlugin**(*config*)
Bases: *BPG.abstract_plugin.AbstractPlugin*

export_content_list(*content_list*, ***kwargs*)
Exports the physical design into the open access format

Parameters **content_list** –

Module contents

BPG.skill package

Submodules

BPG.skill.photonic_skill module

class BPG.skill.photonic_skill.**PhotonicSkillInterface**(*dealer*, *tmp_dir*, *db_config*)
Bases: bag.interface.skill.SkillInterface

create_global_skill_variables(*dataprep_procedure_path*, *dataprep_parameters_path*, *output_file_path*)

Parameters

- **dataprep_procedure_path** –
- **dataprep_parameters_path** –
- **output_file_path** –

dataprep(*lib_name*, *cell_name*, *debug=False*)

manh(*lib_name*, *cell_name*, *debug=False*)

setup_bpg_skill(*output_path*, *dataprep_procedure_path*, *dataprep_parameters_path*, *daprep_skill_function_path*)

Parameters

- **output_path** –
- **dataprep_procedure_path** –
- **dataprep_parameters_path** –
- **dataprep_skill_function_path** –

Module contents

BPG.workspace_setup package

Submodules

BPG.workspace_setup.setup module

BPG.workspace_setup.setup.**copy_files**()

BPG.workspace_setup.setup_submodules module

BPG.workspace_setup.setup_submodules.**add_git_file**(*fname*)

BPG.workspace_setup.setup_submodules.**add_git_submodule**(*module_name*, *url*)

BPG.workspace_setup.setup_submodules.**get_sch_libraries**(*mod_name*, *mod_info*)

BPG.workspace_setup.setup_submodules.**link_submodule**(*repo_path*, *module_name*)

BPG.workspace_setup.setup_submodules.**run_command**(*cmd*)

BPG.workspace_setup.setup_submodules.**run_main**()

BPG.workspace_setup.setup_submodules.**setup_cds_lib**(*module_list*)

BPG.workspace_setup.setup_submodules.**setup_git_submodules**(*module_list*)

BPG.workspace_setup.setup_submodules.**setup_libs_def**(*module_list*)

BPG.workspace_setup.setup_submodules.**setup_python_path**(*module_list*)

BPG.workspace_setup.setup_submodules.**setup_submodule_links**(*module_list*, *repo_path*)

BPG.workspace_setup.setup_submodules.**write_to_file**(*fname*, *lines*)

Module contents

5.1.2 Submodules

5.1.3 BPG.abstract_plugin module

This module contains abstract classes that should be subclassed to create a new plugin to BPG.

Each plugin should have a main interface class that takes a configuration dictionary as input to the __init__(). Then it should expose several methods to perform basic functions needed by the plugin. Examples include exporting a content list to another representation, setting up simulations and testbench scripts, etc.

class BPG.abstract_plugin.**AbstractPlugin**
Bases: object

export_content_list(*content_lists: List[ContentList]*)
This method will take a content list and generate a script that specifies the structure in the desired software package. Ex: for lumerical, export_content_list will generate an lsf file that tells lumerical how to generate the exact same layout

Parameters **content_lists** –

5.1.4 BPG.bpg_custom_types module

5.1.5 BPG.content_list module

This module defines the content list object that is used in db.

class BPG.content_list.**ContentList**(*cell_name: str = ''*, ***kwargs*)
Bases: collections.UserDict

add_item(*key: str*, *value: Any*)
Given a content item and which content type it is, add it to the current ContentList

Parameters

- **key** (*str*) – key corresponding to the content type (rect_list, via_list, etc) key must be in ContentList.all_iterable_keys
- **value** (*Any*) – The content item of the given key type to add to the key list in the current ContentList

all_iterables_keys = ('inst_list', 'rect_list', 'via_list', 'pin_list', 'path_list', '

blockage_list

boundary_list

cell_name

copy()
Copies the shape lists. Does not copy the inst_list (ie the new object will point to the original instance list)

extend_content_list(*new_content: BPG.content_list.ContentList*) → None
Extends the current ContentList object's layout shapes with those of the passed new_content ContentList. Does not extend the instances.

Parameters **new_content** –

get_content_by_layer(*layer: Tuple[str, str]*) → BPG.content_list.ContentList
Return all the shapes in this content list that are on the passed layer. Does not look at instances. Does not via objects.

Parameters **layer** (*Tuple[str, str]*) – The layer purpose pair on which the content of this ContentList will be returned.

Returns **layer_content** – The content of this ContentList that is on the passed layer.

Return type *ContentList*

inst_list

layout_objects_keys = ('rect_list', 'via_list', 'pin_list', 'path_list', 'blockage_lis

monitor_list

path_list

pin_list

polygon_list

rect_list

round_list

sim_list

sort_content_list_by_layers() → Dict[Tuple[str, str], BPG.content_list.ContentList]
Sorts the given content list into a dictionary of content lists, with keys corresponding to a given lpp ASSUMES: the current content list is flat with no via objects

Notes

1) Unpack the content list

2) Loop over objects in the content list, ignoring vias

3) Create new layer dictionary key if object layer is new, and whose value is a content list style array

4) Append object to proper location in the per-layer content list array

source_list

to_bag_tuple_format() → Tuple
Returns the BAG tuple format of the ContentList

Returns **content_tuple** – The content of the ContentList object in the BAG tuple format

Return type Tuple

transform_content(*res: float, loc: Tuple[Union[float, int], Union[float, int]], orient: str, via_info: Dict[KT, VT], unit_mode: bool*) → BPG.content_list.ContentList

Transforms the layout content (does not transform the sub-instances) of the current ContentList by the loc and orient passed.

Parameters

- **res** (*float*) – The grid resolution.
- **loc** (*Tuple[Union[float, int], Union[float, int]]*) – The (x, y) tuple describing the translation vector for the transformation.
- **orient** (*str*) – The orientation string describing how the layout should be rotated.
- **via_info** (*Dict*) – A dictionary containing the via technology properties
- **unit_mode** (*bool*) – True if loc is provided in resolution unit coordinates. False if in layout unit coordinates.

Returns **new_content_list** – The new ContentList object with the transformed shapes.

Return type *ContentList*

via_list

via_to_polygon_and_delete(*via_info: Dict[KT, VT]*)

static via_to_polygon_list(*via: PhotonicViaInfo, via_lay_info: Dict[KT, VT], x0: Union[float, int], y0: Union[float, int]*) → List[T]

5.1.6 BPG.db module

class BPG.db.**PhotonicTemplateDB**(*lib_defs: str, routing_grid: RoutingGrid, lib_name: str, prj: Optional[BagProject] = None, use_cybagoa: bool = False, gds_lay_file: str = '', photonic_tech_info: PhotonicTechInfo = None, **kwargs*)

Bases: bag.layout.template.TemplateDB

dataprep(*flat_content_list: BPG.content_list.ContentList, is_lsf: bool = False*) → BPG.content_list.ContentList

Initializes the dataprep plugin with the standard tech info and runs the dataprep procedure

Parameters

- **flat_content_list** (ContentList) – The flattened Contentlist of the master
- **is_lsf** (*bool*) – True if running LSF dataprep. False if running standard dataprep.

Returns **post_dataprep_flat_content_list** – The ContentList object (no longer layer separated) after running dataprep

Return type *ContentList*

generate_content_list(*master_list: Sequence[DesignMaster], name_list: Optional[Sequence[Optional[str]]] = None, lib_name: str = '', rename_dict: Optional[Dict[str, str]] = None*) → List[BPG.content_list.ContentList]

Create the content list from the provided masters and returns it.

Parameters

- **master_list** (*Sequence[DesignMaster]*) – list of masters to instantiate.
- **name_list** (*Optional[Sequence[Optional[str]]]*) – list of master cell names. If not given, default names will be used.

- **lib_name** (*str*) – Library to create the masters in. If empty or None, use default library.
- **rename_dict** (*Optional[Dict[str, str]]*) – optional master cell renaming dictionary.

Returns **content_list** – Generated content list of the provided masters

Return type List[*ContentList*]

generate_flat_content_list(*master_list: Sequence[PhotonicTemplateBase], name_list: Optional[Sequence[Optional[str]]] = None, lib_name: str = '', rename_dict: Optional[Dict[str, str]] = None*) → List[ContentList]

Create all given masters in the database to a flat hierarchy.

Parameters

- **master_list** (*Sequence[DesignMaster]*) – list of masters to instantiate.
- **name_list** (*Optional[Sequence[Optional[str]]]*) – list of master cell names. If not given, default names will be used.
- **lib_name** (*str*) – Library to create the masters in. If empty or None, use default library.
- **rename_dict** (*Optional[Dict[str, str]]*) – optional master cell renaming dictionary.

5.1.7 BPG.geometry module

class BPG.geometry.**Box**

Bases: object

A class representing a 3D rectangle

align_to_port(*port*, *offset=(0, 0)*)

Moves the center of the simulation object to align to the provided photonic port

Parameters

- **port** (PhotonicPort) – Photonic port for the simulation object to be aligned to
- **offset** (*Tuple*) – (x, y) offset relative to the port location

move_by(*dx*, *dy*, *unit_mode=False*)

set_center_span(*dim*, *center*, *span*)

Sets the center and span of a given geometry dimension

Parameters

- **dim** (*str*) – 'x', 'y', or 'z' for the corresponding dimension
- **center** (*float*) – coordinate location of the center of the geometry
- **span** (*float*) – length of the geometry along the dimension

set_span(*dim*, *span*)

Sets the span of a given geometry dimension

Parameters

- **dim** (*str*) – 'x', 'y', or 'z' for the corresponding dimension
- **span** (*float*) – length of the geometry along the dimension

class BPG.geometry.**CoordBase**
Bases: object

A class representing the basic unit of measurement for all objects in BPG.

All user-facing values are assumed to be floating point numbers in units of microns. BAG internal functions assume that we receive 'unit-mode' numbers, which are integers in units of nanometers. Both formats are supported.

float
Returns the rounded floating point number closest to a valid point on the resolution grid

meters
Returns the rounded floating point number in meters closest to a valid point on the resolution grid

micron = Decimal('0.000001')

microns

res = Decimal('0.001')

unit_mode

value

class BPG.geometry.**Plane**
Bases: object

A class representing a plane that is orthogonal to one of the cardinal axes

TODO: Implement this class

class BPG.geometry.**XY**
Bases: object

A class representing a single point on the XY plane

x

x_float

x_meters

xy

xy_float

xy_meters

y

y_float

y_meters

class BPG.geometry.**XYZ**
Bases: object

A class representing a single point on the XYZ space

x

x_float

x_meters

xyz

xyz_float

xyz_meters

y

y_float

y_meters

z

z_float

z_meters

5.1.8 BPG.layout_manager module

class BPG.layout_manager.**PhotonicLayoutManager**(*spec_file: str*, *bag_config_path: str = None*, *port: int = None*)

Bases: *BPG.photonic_core.PhotonicBagProject*

User-facing class that enables encapsulated dispatch of layout operations such as generating gds, oa, lsf, etc

create_materials_file()
Takes the custom materials stated in the lumerical_map and generates a Lumerical lsf file that defines the materials for use in simulation.

dataprep()
Performs dataprep on the design

generate_content(*layout_params: dict = None*, *cell_name: str = None*) → List[ContentList]
Generates a content list. If layout params and cell name are passed, use these parameters. If not provided, get these variables from the spec file

Parameters

- **layout_params** (*dict*) – Optional dictionary of parameters to be sent to the layout generator class.
- **cell_name** (*str*) – Optional name of the cell to be created from the layout generator.

Returns **content_list** – A db of all generated shapes

Return type *ContentList*

generate_dataprep_gds() → None
Exports the dataprep content to GDS format

generate_flat_content() → ContentList
Generates a flattened content list from the template passed to generate_content.

Returns **content_list** – A db of all generated shapes

Return type *ContentList*

generate_flat_gds() → None
Exports flattened content list of design to gds format

generate_gds() → None
Exports the content list to gds format

generate_lsf(*create_materials=True*)
Converts generated layout to lsf format for lumerical import

generate_tb(*debug=False*)
Generates the lumerical testbench lsf

init_plugins() → None
Creates all built-in plugins based on the provided configuration and tech-info

load_content_list(*content_list: str*, *filepath: str = None*)
Loads the specified content list from the passed filepath, or the PLM default filepath.

Parameters

- **content_list** (*str*) – Which content list to load
- **filepath** (*Optional[str]*) – Filepath (directory and filename) (relative to where bpg was started) to which content list file to load. If not specified, defaults to the directory provided in the current PhotonicLayoutManager instance.

save_content_list(*content_list: str*, *filepath: str = None*)
Saves the provided content list to the passed filepath, or to the PLM default filepath.

Parameters

- **content_list** (*str*) – Which content list to save
- **filepath** (*Optional[str]*) – Filepath (directory and filename) (relative to where bpg was started) where to store the file. If not specified, defaults to the directory provided in the current PhotonicLayoutManager instance.

5.1.9 BPG.logger module

BPG.logger.**setup_logger**(*log_path: str*, *log_filename: str = 'bpg.log'*) → None
Configures the root logger so that all other loggers in BPG inherit from its properties.

Parameters

- **log_path** (*str*) – The path to save the log files.
- **log_filename** (*str*) – The name of the primary output log file.

5.1.10 BPG.objects module

This module defines various layout objects one can add and manipulate in a template.

class BPG.objects.**PhotonicAdvancedPolygon**(*resolution: float, layer: Union[str, Tuple[str, str]], points: List[Tuple[Union[float, int], Union[float, int]]], negative_points: Union[List[Tuple[Union[float, int], Union[float, int]]], List[List[Tuple[Union[float, int], Union[float, int]]]]], unit_mode: bool = False*)
Bases: bag.layout.objects.Polygon

A layout polygon object.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **layer** (*Union[str, Tuple[str, str]]*) – the layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – the points defining the polygon.

- **unit_mode** (*bool*) – True if the points are given in resolution units.

class BPG.objects.**PhotonicBlockage**(*resolution*, *block_type*, *block_layer*, *points*, *unit_mode=False*)

Bases: bag.layout.objects.Blockage

A blockage object.

Subclass Polygon for code reuse.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **block_type** (*str*) – the blockage type. Currently supports 'routing' and 'placement'.
- **block_layer** (*str*) – the blockage layer. This value is ignored if blockage type is 'placement'.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – the points defining the blockage.
- **unit_mode** (*bool*) – True if the points are given in resolution units.

classmethod from_content(*content*, *resolution*)

class BPG.objects.**PhotonicBoundary**(*resolution: float, boundary_type: str, points: List[Tuple[Union[float, int], Union[float, int]]], unit_mode: bool = False*)

Bases: bag.layout.objects.Boundary

A boundary object.

Subclass Polygon for code reuse.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **boundary_type** (*str*) – the boundary type. Currently supports 'PR', 'snap', and 'area'.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – the points defining the blockage.
- **unit_mode** (*bool*) – True if the points are given in resolution units.

classmethod from_content(*content*, *resolution*)

class BPG.objects.**PhotonicInstance**(*parent_grid: bag.layout.routing.grid.RoutingGrid, lib_name: str, master, loc: Tuple[Union[float, int], Union[float, int]], orient: str, name: str = None, nx: int = 1, ny: int = 1, spx: int = 0, spy: int = 0, unit_mode: bool = False*)

Bases: bag.layout.objects.Instance

A photonic layout instance, with optional arraying parameters. This class adds the ability to read

Parameters

- **parent_grid** (*RoutingGrid*) – the parent RoutingGrid object.
- **lib_name** (*str*) – the layout library name.
- **master** (*TemplateBase*) – the master template of this instance.
- **loc** (*Tuple[Union[float, int], Union[float, int]]*) – the origin of this instance.

- **orient** (*str*) – the orientation of this instance.
- **name** (*Optional[str]*) – name of this instance.
- **nx** (*int*) – number of columns.
- **ny** (*int*) – number of rows.
- **spx** (*Union[float, int]*) – column pitch.
- **spy** (*Union[float, int]*) – row pitch.
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units.

content
A dictionary representation of this instance.

get_bound_box_of(*row: int = 0*, *col: int = 0*) → bag.layout.util.BBox
Returns the bounding box of an instance in this mosaic.

get_photonic_port(*name: str*, *row: int = 0*, *col: int = 0*) → PhotonicPort
Returns the photonic port object associated with the provided port name

Parameters

- **name** (*str*) – name of the port to be returned
- **row** (*int*) – row in the array of instances to be accessed
- **col** (*int*) – column in the array of instances to be accessed

Returns **port** – photonic port object associated with the provided name

Return type *PhotonicPort*

get_port_used(*port_name: str*) → bool

master
The master template of this instance.

move_by(*dx: Union[float, int] = 0*, *dy: Union[float, int] = 0*, *unit_mode: bool = False*) → None
Move this instance by the given amount.

Parameters

- **dx** (*Union[float, int]*) – the X shift.
- **dy** (*Union[float, int]*) – the Y shift.
- **unit_mode** (*bool*) – True if shifts are given in resolution units

set_port_used(*port_name: str*) → None

transform(*loc: Tuple[Union[float, int], Union[float, int]] = (0, 0)*, *orient: str = 'R0'*, *unit_mode: bool = False*, *copy: bool = False*) → Optional[bag.layout.objects.Figure]
Transform this figure.

class BPG.objects.**PhotonicInstanceInfo**(*res*, *change_orient=True*, ***kwargs*)
Bases: bag.layout.objects.InstanceInfo

A dictionary that represents a layout instance.

content()

copy()
Override copy method of InstanceInfo to return a PhotonicInstanceInfo instead.

master_key

param_list = ['lib', 'cell', 'view', 'name', 'loc', 'orient', 'num_rows', 'num_cols',

class BPG.objects.**PhotonicPath**(*resolution: float, layer: Union[str, Tuple[str, str]], width: Union[float, int], points: List[Tuple[Union[float, int], Union[float, int]]], unit_mode: bool = False*)

Bases: bag.layout.objects.Figure

A path defined by a list of x,y points and a width.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **layer** (*string or (string, string)*) – the layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **width** (*float*) – width of this path, in layout units.
- **points** (*List[Tuple[float, float]]*) – list of path points.
- **unit_mode** (*bool*) – True if width and points are given as resolution units instead of layout units.

content

A dictionary representation of this path.

classmethod from_content(*content*, *resolution*)

layer

The rectangle (layer, purpose) pair.

move_by(*dx: Union[float, int] = 0*, *dy: Union[float, int] = 0*, *unit_mode: bool = False*) → None

Move this path by the given amount.

Parameters

- **dx** (*float*) – the X shift.
- **dy** (*float*) – the Y shift.
- **unit_mode** (*bool*) – True if shifts are given in resolution units.

points

Return non-unit mode python list of points

classmethod polygon_pointlist_export(*vertices: List[Tuple[float, float]]*) → Tuple[List[T], List[T]]

Parameters **vertices** (*List[Tuple[float, float]]*) – The verticies from the content list of this polygon

Returns **output_list** – The positive and negative polygon pointlists describing this polygon

Return type Tuple[List, List]

polygon_points

Return non-unit_mode python list of points

static process_points(*points: numpy.ndarray, width: Union[float, int], eps: float = 0.0001*) → numpy.ndarray

Takes a width and center point list, and returns the points describing the polygon of the path.

While the center points are not rounded to a grid, the points describing the polygon are snapped to the layout grid.

The returned polygon points are in unit_mode (grid resolution units).

Parameters

- **points** (*np.ndarray*) – A numpy array of points describing the center of the path.
- **width** (*Union[int, float]*) – The width of the path
- **eps** (*float*) – The tolerance for determining whether two points are coincident

Returns polygon_points – The numpy array of points describing the polygon of the path, rounded to the layout grid, and expressed in resolution units (ints).

Return type np.ndarray

transform(*loc: Tuple[Union[float, int], Union[float, int]] = (0, 0), orient: str = 'R0', unit_mode: bool = False, copy: bool = False*) → bag.layout.objects.Figure

Transform this figure.

valid

Returns True if this instance is valid.

width

width_unit

class BPG.objects.**PhotonicPathCollection**(*resolution*, *paths*)

Bases: bag.layout.objects.PathCollection

A layout figure that consists of one or more paths.

This class make it easy to draw bus/trasmission line objects.

Parameters

- **resolution** (*float*) – layout unit resolution.
- **paths** (*List[Path]*) – paths in this collection.

class BPG.objects.**PhotonicPinInfo**(*res*, ***kwargs*)

Bases: bag.layout.objects.PinInfo

A dictionary that represents a layout pin.

classmethod from_content(*content: Dict[KT, VT], resolution: float*) → BPG.objects.PhotonicPinInfo

param_list = ['net_name', 'pin_name', 'label', 'layer', 'bbox', 'make_rect']

transform(*loc: Tuple[Union[float, int], Union[float, int]], orient: str, unit_mode: bool, copy: bool*)

class BPG.objects.**PhotonicPolygon**(*resolution: float, layer: Union[str, Tuple[str, str]], points: List[Tuple[Union[float, int], Union[float, int]]], unit_mode: bool = False*)

Bases: bag.layout.objects.Polygon

A layout polygon object.

Parameters

- **resolution** (*float*) – the layout grid resolution.
- **layer** (*Union[str, Tuple[str, str]]*) – the layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – the points defining the polygon.
- **unit_mode** (*bool*) – True if the points are given in resolution units.

classmethod from_content(*content*, *resolution*)

classmethod lsf_export(*vertices*, *layer_prop*) → List[str]
Describes the current polygon shape in terms of lsf parameters for lumerical use

Parameters

- **vertices** (*List[Tuple[float, float]]*) – ordered list of x,y coordinates representing the points of the polygon
- **layer_prop** (*dict*) – dictionary containing material properties for the desired layer

Returns **lsf_code** – list of str containing the lsf code required to create specified rectangles

Return type List[str]

classmethod polygon_pointlist_export(*vertices: List[Tuple[float, float]]*) → Tuple[List[T], List[T]]

Parameters **vertices** (*List[Tuple[float, float]*) – The verticies from the content list of this polygon

Returns **output_list** – The positive and negative polygon pointlists describing this polygon

Return type Tuple[List, List]

class BPG.objects.**PhotonicRect**(*layer*, *bbox*, *nx=1*, *ny=1*, *spx=0*, *spy=0*, *unit_mode=False*)
Bases: bag.layout.objects.Rect

A layout rectangle, with optional arraying parameters.

Parameters

- **layer** (*string or (string, string)*) – the layer name, or a tuple of layer name and purpose name. If pupose name not given, defaults to 'drawing'.
- **bbox** (*bag.layout.util.BBox or bag.layout.util.BBoxArray*) – the base bounding box. If this is a BBoxArray, the BBoxArray's arraying parameters are used.
- **nx** (*int*) – number of columns.
- **ny** (*int*) – number of rows.
- **spx** (*float*) – column pitch.
- **spy** (*float*) – row pitch.
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units.

classmethod from_content(*content*, *resolution*)

classmethod lsf_export(*bbox*, *layer_prop*, *nx=1*, *ny=1*, *spx=0.0*, *spy=0.0*) → List[str]
Describes the current rectangle shape in terms of lsf parameters for lumerical use. Note that Lumerical uses meters as the base unit, and all input coords are assumed to be in microns. This method inherently resizes

Parameters

- **bbox** (*[[float, float], [float, float]]*) – lower left and upper right corner xy coordinates
- **layer_prop** (*dict*) – dictionary containing material properties for the desired layer
- **nx** (*int*) – number of arrayed rectangles in the x-direction
- **ny** (*int*) – number of arrayed rectangles in the y-direction
- **spx** (*float*) – space between arrayed rectangles in the x-direction
- **spy** (*float*) – space between arrayed rectangles in the y-direction

Returns **lsf_code** – list of str containing the lsf code required to create specified rectangles

Return type List[str]

classmethod polygon_pointlist_export(*bbox: [[<class 'int'>, <class 'int'>], [<class 'int'>, <class 'int'>]], nx: int = 1, ny: int = 1, spx: Union[float, int] = 0.0, spy: Union[float, int] = 0.0*) → Tuple[List[T], List[T]]

Convert the PhotonicRect geometry to a list of polygon pointlists.

Parameters

- **bbox** (*[[float, float], [float, float]]*) – lower left and upper right corner xy coordinates
- **nx** (*int*) – number of arrayed rectangles in the x-direction
- **ny** (*int*) – number of arrayed rectangles in the y-direction
- **spx** (*float*) – space between arrayed rectangles in the x-direction
- **spy** (*float*) – space between arrayed rectangles in the y-direction

Returns **output_list** – The positive and negative polygon pointlists describing the photonicRect

Return type Tuple[List, List]

transform(*loc: Tuple[Union[float, int], Union[float, int]] = (0, 0), orient: str = 'R0', unit_mode: bool = False, copy: bool = False*) → Optional[bag.layout.objects.Figure]

Transform this figure.

class BPG.objects.**PhotonicRound**(*layer: Union[str, Tuple[str, str]], resolution: float, rout: Union[float, int], center: Tuple[Union[float, int], Union[float, int]] = (0, 0), rin: Union[float, int] = 0, theta0: Union[float, int] = 0, theta1: Union[float, int] = 360, nx: int = 1, ny: int = 1, spx: Union[float, int] = 0, spy: Union[float, int] = 0, unit_mode: bool = False*)

Bases: bag.layout.objects.Arrayable

A layout round object, with optional arraying parameters.

Parameters

- **layer** (*string or (string, string)*) – the layer name, or a tuple of layer name and purpose name. If pupose name not given, defaults to 'drawing'.
- **rout** –
- **rin** –
- **theta0** –
- **theta1** –
- **nx** (*int*) – number of columns.
- **ny** (*int*) – number of rows.
- **spx** (*float*) – column pitch.
- **spy** (*float*) – row pitch.
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units.

center

The center in layout units

center_unit
The center in resolution units

content
A dictionary representation of this rectangle.

classmethod from_content(*content*, *resolution*)

layer
The rectangle (layer, purpose) pair.

classmethod lsf_export(*rout: Union[float, int], rin: Union[float, int], theta0: Union[float, int], theta1: Union[float, int], layer_prop: Dict[KT, VT], center: Tuple[Union[float, int], Union[float, int]], nx: int = 1, ny: int = 1, spx: Union[float, int] = 0.0, spy: Union[float, int] = 0.0*) → List[str]

Parameters

- **rout** –
- **rin** –
- **theta0** –
- **theta1** –
- **layer_prop** –
- **center** –
- **nx** –
- **ny** –
- **spx** –
- **spy** –

move_by(*dx: Union[float, int] = 0, dy: Union[float, int] = 0, unit_mode: bool = False*)
Moves the round object

static num_of_sparse_point_round(*radius: float, res_grid_size: float*) → int

classmethod polygon_pointlist_export(*rout: Union[float, int], rin: Union[float, int], theta0: Union[float, int], theta1: Union[float, int], center: Tuple[Union[float, int], Union[float, int]], nx: int = 1, ny: int = 1, spx: Union[float, int] = 0.0, spy: Union[float, int] = 0.0, resolution: float = 0.001*)

rin
The inner radius in layout units

rin_unit
The inner radius in resolution units

rout
The outer radius in layout units

rout_unit
The outer radius in resolution units

theta0
The starting angle, in degrees

theta1
The ending angle, in degrees

transform(*loc: Tuple[Union[float, int], Union[float, int]] = (0, 0), orient: str = 'R0', unit_mode: bool = False, copy: bool = False*) → Optional[BPG.objects.PhotonicRound]
Transform this figure.

class BPG.objects.**PhotonicTLineBus**(*resolution*, *layer*, *points*, *widths*, *spaces*, *end_style='truncate'*, *unit_mode=False*)
Bases: bag.layout.objects.TLineBus

A transmission line bus drawn using Path.

assumes only 45 degree turns are used, and begin and end line segments are straight.

Parameters

- **resolution** (*float*) – layout unit resolution.
- **layer** (*Union[str, Tuple[str, str]]*) – the bus layer.
- **points** (*List[Tuple[Union[float, int], Union[float, int]]]*) – list of center points of the bus.
- **widths** (*List[Union[float, int]]*) – list of wire widths. 0 index is left/bottom most wire.
- **spaces** (*List[Union[float, int]]*) – list of wire spacings.
- **end_style** (*str*) – the path ends style. Currently support 'truncate', 'extend', and 'round'.
- **unit_mode** (*bool*) – True if width and points are given as resolution units instead of layout units.

class BPG.objects.**PhotonicVia**(*tech*, *bbox*, *bot_layer*, *top_layer*, *bot_dir*, *nx=1*, *ny=1*, *spx=0*, *spy=0*, *extend=True*, *top_dir=None*, *unit_mode=False*)
Bases: bag.layout.objects.Via

A layout via, with optional arraying parameters.

Parameters

- **tech** (*bag.layout.core.TechInfo*) – the technology class used to calculate via information.
- **bbox** (*bag.layout.util.BBox or bag.layout.util.BBoxArray*) – the via bounding box, not including extensions. If this is a BBoxArray, the BBoxArray's arraying parameters are used.
- **bot_layer** (*str or (str, str)*) – the bottom layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **top_layer** (*str or (str, str)*) – the top layer name, or a tuple of layer name and purpose name. If purpose name not given, defaults to 'drawing'.
- **bot_dir** (*str*) – the bottom layer extension direction. Either 'x' or 'y'.
- **nx** (*int*) – arraying parameter. Number of columns.
- **ny** (*int*) – arraying parameter. Mumber of rows.
- **spx** (*float*) – arraying parameter. Column pitch.
- **spy** (*float*) – arraying parameter. Row pitch.
- **extend** (*bool*) – True if via extension can be drawn outside of bounding box.

- **top_dir** (*Optional[str]*) – top layer extension direction. Can force to extend in same direction as bottom.
- **unit_mode** (*bool*) – True if array pitches are given in resolution units.

classmethod from_content(*content: Dict[KT, VT]*) → BPG.objects.PhotonicVia

class BPG.objects.**PhotonicViaInfo**(*res*, ***kwargs*)

Bases: bag.layout.objects.ViaInfo

A dictionary that represents a layout via.

param_list = ['id', 'loc', 'orient', 'num_rows', 'num_cols', 'sp_rows', 'sp_cols', 'en

5.1.11 BPG.photonic_core module

class BPG.photonic_core.**DummyPhotonicTechInfo**(*photonic_tech_params*, *resolution*, *layout_unit*)

Bases: *BPG.photonic_core.PhotonicTechInfo*

A dummy PhotonicTechInfo class

height(*layer: Union[str, Tuple[str, str]]*)

Returns the height from the top of the silicon region (defined as 0) to the bottom surface of the given layer, in layout units.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **height** – The height of the bottom surface for shapes on the layer

Return type float

height_unit(*layer: Union[str, Tuple[str, str]]*)

Returns the height from the top of the silicon region (defined as 0) to the bottom surface of the given layer, in resolution units.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **height_unit** – The height of the bottom surface in resolution units for shapes on the layer

Return type float

max_width(*layer: Union[str, Tuple[str, str]]*)

Returns the maximum width (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **max_width** – The maximum width for shapes on the layer

Return type float

max_width_unit(*layer: Union[str, Tuple[str, str]]*)

Returns the maximum width (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **max_width_unit** – The maximum width in resolution units for shapes on the layer

Return type float

min_area (*layer: Union[str, Tuple[str, str]]*)
Returns the minimum area (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_area** – The minimum area for shapes on the layer

Return type float

min_area_unit (*layer: Union[str, Tuple[str, str]]*)
Returns the minimum area (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_area_unit** – The minimum area in resolution units for shapes on the layer

Return type float

min_edge_length (*layer: Union[str, Tuple[str, str]]*)
Returns the minimum edge length (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_edge_length** – The minimum edge length for shapes on the layer

Return type float

min_edge_length_unit (*layer: Union[str, Tuple[str, str]]*)
Returns the minimum edge length (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_edge_length** – The minimum edge length in resolution units for shapes on the layer

Return type float

min_space (*layer: Union[str, Tuple[str, str]]*)
Returns the minimum space (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_space** – The minimum space for shapes on the layer

Return type float

min_space_unit (*layer: Union[str, Tuple[str, str]]*)
Returns the minimum space (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_space_unit** – The minimum space in resolution units for shapes on the layer

Return type float

min_width (*layer: Union[str, Tuple[str, str]]*)
Returns the minimum width (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_width** – The minimum width for shapes on the layer

Return type float

min_width_unit (*layer: Union[str, Tuple[str, str]]*)
Returns the minimum width (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_width_unit** – The minimum width in resolution units for shapes on the layer

Return type float

sheet_resistance (*layer: Union[str, Tuple[str, str]]*) → float
Returns the sheet resistance of the layer, in Ohm/sq.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **rs** – The sheet resistance of the layer in Ohm/sq

Return type float

thickness (*layer: Union[str, Tuple[str, str]]*)
Returns the thickness of the layer, in layout units.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **thickness** – The thickness of shapes on the layer

Return type float

thickness_unit (*layer: Union[str, Tuple[str, str]]*)
Returns the thickness of the layer, in resolution units

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **thickness_unit** – The thickness in resolution units for shapes on the layer

Return type float

class BPG.photonic_core.**PhotonicBagLayout** (*grid*, *use_cybagoa=False*)
Bases: bag.layout.core.BagLayout

This class contains layout information of a cell.

Parameters

- **grid** (bag.layout.routing.RoutingGrid) – the routing grid instance.
- **use_cybagoa** (*bool*) – True to use cybagoa package to accelerate layout.

add_monitor_obj (*monitor_obj*)
Add a new Lumerical monitor object to the db

add_path (*path: PhotonicPath*)
Add a new path.

Parameters **path** (*Path*) – the path object to add.

add_round (*round_obj: PhotonicRound*)
Add a new (arrayed) round shape.

Parameters **round_obj** (BPG.objects.PhotonicRound) – the round object to add.

add_sim_obj(*sim_obj*)
Add a new Lumerical simulation object to the db

add_source_obj(*source_obj*)
Add a new Lumerical source object to the db

finalize()
Prevents any further changes to this layout.

get_content(*lib_name: str, cell_name: str, rename_fun: Callable[[str], str]*) → Union[BPG.content_list.ContentList, Tuple[str, cybagoa.PyOALayout]]
Returns a list describing geometries in this layout.

Parameters

- **lib_name** (*str*) – the layout library name.
- **cell_name** (*str*) – the layout top level cell name.
- **rename_fun** (*Callable[[str], str]*) – the layout cell renaming function.

Returns **content** – a ContentList describing this layout, or PyOALayout if cybagoa package is enabled.

Return type Union[*ContentList*, Tuple[str, 'cybagoa.PyOALayout']]

move_all_by(*dx: Union[float, int] = 0.0*, *dy: Union[float, int] = 0.0*, *unit_mode: bool = False*) → None
Move all layout objects in this layout by the given amount.

Parameters

- **dx** (*Union[float, int]*) – the X shift.
- **dy** (*Union[float, int]*) – the Y shift.
- **unit_mode** (*bool*) – True if shift values are given in resolution units.

class BPG.photonic_core.**PhotonicBagProject**(*bag_config_path=None*, *port=None*)
Bases: bag.core.BagProject

The main bag controller class.

This class extracts user configuration variables and issues high level bag commands. Most config variables have defaults pointing to files in the BPG/examples/tech folder

Parameters

- **bag_config_path** (*Optional[str]*) – the bag configuration file path. If None, will attempt to read from environment variable BAG_CONFIG_PATH.
- **port** (*Optional[int]*) – the BAG server process port number. If not given, will read from port file.

load_spec_file_paths(*spec_file*)
Receives a specification file from the user and configures the project paths accordingly

static load_yaml(*filepath*)
Setup standardized method for yaml loading

class BPG.photonic_core.**PhotonicTechInfo**(*photonic_tech_params*, *resolution*, *layout_unit*)
Bases: object

height(*layer: Union[str, Tuple[str, str]]*) → float
Returns the height from the top of the silicon region (defined as 0) to the bottom surface of the given layer, in layout units.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **height** – The height of the bottom surface for shapes on the layer

Return type float

height_unit (*layer: Union[str, Tuple[str, str]]*) → int
Returns the height from the top of the silicon region (defined as 0) to the bottom surface of the given layer, in resolution units.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **height_unit** – The height of the bottom surface in resolution units for shapes on the layer

Return type float

load_tech_files ()

max_width (*layer: Union[str, Tuple[str, str]]*) → float
Returns the maximum width (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **max_width** – The maximum width for shapes on the layer

Return type float

max_width_unit (*layer: Union[str, Tuple[str, str]]*) → int
Returns the maximum width (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **max_width_unit** – The maximum width in resolution units for shapes on the layer

Return type float

min_area (*layer: Union[str, Tuple[str, str]]*) → float
Returns the minimum area (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_area** – The minimum area for shapes on the layer

Return type float

min_area_unit (*layer: Union[str, Tuple[str, str]]*) → int
Returns the minimum area (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_area_unit** – The minimum area in resolution units for shapes on the layer

Return type float

min_edge_length (*layer: Union[str, Tuple[str, str]]*) → float
Returns the minimum edge length (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_edge_length** – The minimum edge length for shapes on the layer

Return type float

min_edge_length_unit (*layer: Union[str, Tuple[str, str]]*) → int
Returns the minimum edge length (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_edge_length** – The minimum edge length in resolution units for shapes on the layer

Return type float

min_space (*layer: Union[str, Tuple[str, str]]*) → float
Returns the minimum space (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_space** – The minimum space for shapes on the layer

Return type float

min_space_unit (*layer: Union[str, Tuple[str, str]]*) → int
Returns the minimum space (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_space_unit** – The minimum space in resolution units for shapes on the layer

Return type float

min_width (*layer: Union[str, Tuple[str, str]]*) → float
Returns the minimum width (in layout units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_width** – The minimum width for shapes on the layer

Return type float

min_width_unit (*layer: Union[str, Tuple[str, str]]*) → int
Returns the minimum width (in resolution units) for a given layer.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **min_width_unit** – The minimum width in resolution units for shapes on the layer

Return type float

sheet_resistance (*layer: Union[str, Tuple[str, str]]*) → float
Returns the sheet resistance of the layer, in Ohm/sq.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **rs** – The sheet resistance of the layer in Ohm/sq

Return type float

thickness (*layer: Union[str, Tuple[str, str]]*) → float
Returns the thickness of the layer, in layout units.

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **thickness** – The thickness of shapes on the layer

Return type float

thickness_unit(*layer: Union[str, Tuple[str, str]]*) → int
Returns the thickness of the layer, in resolution units

Parameters **layer** (*Union[str, Tuple[str, str]]*) – The layer name or LPP of the layer.

Returns **thickness_unit** – The thickness in resolution units for shapes on the layer

Return type float

BPG.photonic_core.**create_photonic_tech_info**(*bpg_config: Dict[KT, VT], tech_info: TechInfo*) → PhotonicTechInfo
Create PhotonicTechInfo object.

5.1.12 BPG.port module

class BPG.port.**PhotonicPort**(*name: str*, *center: coord_type*, *orient: str*, *width: dim_type*, *layer: lpp_type*, *resolution: float*, *unit_mode: bool = False*)
Bases: object

center
Return the center coordinates as np array

center_unit
Return the center coordinates as np array in resolution units

classmethod from_dict(*center: coord_type*, *name: str*, *orient: str*, *port_width: dim_type*, *layer: layer_or_lpp_type*, *resolution: float*, *unit_mode: bool = True*) → PhotonicPort
Creates a new PhotonicPort object from a set of arguments

Parameters

- **center** (*Tuple[Union[float, int], Union[float, int]]*) – the (x, y) point of the port
- **name** (*str*) – the name of the port
- **orient** (*str*) – the orientation pointing into the object of the port
- **port_width** (*Union[float, int]*) – the port width
- **layer** (*Union[Tuple[str, str], str]*) – the layer / layer purpose pair on which the port should be drawn. If the purpose is not specified, it is defaulted to the 'port' purpose
- **resolution** (*float*) – the grid resolution
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units

Returns **port** – the generated port

Return type *PhotonicPort*

is_horizontal() → bool
Returns True if port orientation is pointing in a horizontal direction

is_vertical() → bool
Returns True if port orientation is in a vertical direction

layer
Returns the layer of the port

name
Returns the name of the port

orientation
Returns the orientation of the port

resolution
Returns the layout resolution of the port object

transform(*loc: coord_type = (0, 0)*, *orient: str = 'R0'*, *unit_mode: bool = False*) → PhotonicPort
Return a new transformed photonic port

Parameters

- **loc** (*Tuple[Union[float, int], Union[float, int]]*) – the x, y coordinate to move the port
- **orient** (*str*) – the orientation to rotate the port
- **unit_mode** (*bool*) – true if layout dimensions are specified in resolution units

Returns **port** – the transformed photonic port object

Return type *PhotonicPort*

used
Returns True if port is used

width
Returns the width of the port

width_unit
Returns the width of the port in layout units

width_vec(*unit_mode: bool = True*, *normalized: bool = True*) → numpy.core.multiarray.array
Returns a normalized vector pointing into the port object

Parameters

- **unit_mode** (*bool*) – True to return vector in resolution units
- **normalized** (*bool*) – True to normalize the vector. If False, vector magnitude is the port width

Returns **vec** – a vector whos orientation points into the port and whos magnitude is either 1 or the waveguide port width

Return type np.array

5.1.13 BPG.template module

class BPG.template.**PhotonicTemplateBase**(*temp_db: PhotonicTemplateDB*, *lib_name: str*, *params: Dict[str, Any]*, *used_names: Set[str]*, ***kwargs*)
Bases: bag.layout.template.TemplateBase

add_instance(*master: BPG.template.PhotonicTemplateBase, inst_name: Optional[str] = None, loc: Tuple[Union[float, int], Union[float, int]] = (0, 0), orient: str = 'R0', nx: int = 1, ny: int = 1, spx: Union[float, int] = 0, spy: Union[float, int] = 0, unit_mode: bool = False*) → BPG.objects.PhotonicInstance

Adds a new (arrayed) instance to layout.

Parameters

- **master** (*TemplateBase*) – the master template object.
- **inst_name** (*Optional[str]*) – instance name. If None or an instance with this name already exists, a generated unique name is used.
- **loc** (*Tuple[Union[float, int], Union[float, int]]*) – instance location.
- **orient** (*str*) – instance orientation. Defaults to "R0"
- **nx** (*int*) – number of columns. Must be positive integer.
- **ny** (*int*) – number of rows. Must be positive integer.
- **spx** (*Union[float, int]*) – column pitch. Used for arraying given instance.
- **spy** (*Union[float, int]*) – row pitch. Used for arraying given instance.
- **unit_mode** (*bool*) – True if dimensions are given in resolution units.

Returns **inst** – the added instance.

Return type *PhotonicInstance*

add_instances_port_to_port(*inst_master: BPG.template.PhotonicTemplateBase, instance_port_name: str, self_port: Optional[BPG.port.PhotonicPort] = None, self_port_name: Optional[str] = None, instance_name: Optional[str] = None, reflect: bool = False*) → BPG.objects.PhotonicInstance

Instantiates a new instance of the inst_master template. The new instance is placed such that its port named 'instance_port_name' is aligned-with and touching the 'self_port' or 'self_port_name' port of the current hierarchy level.

The new instance is rotated about the new instance's master's origin until desired port is aligned. Optional reflection is performed after rotation, about the port axis.

The self port being connected to can be specified either by passing a self_port PhotonicPort object, or by passing the self_port_name, which refers to a port that must exist in the current hierarchy level.

Parameters

- **inst_master** (PhotonicTemplateBase) – the template master to be added
- **instance_port_name** (*str*) – the name of the port in the added instance to connect to
- **self_port** (*Optional[PhotonicPort]*) – the photonic port object in the current hierarchy to connect to. Has priority over self_port_name
- **self_port_name** (*Optional[str]*) – the name of the port in the current hierarchy to connect to
- **instance_name** (*Optional[str]*) – the name to give the new instance
- **reflect** (*bool*) – True to flip the added instance after rotation

Returns **new_inst** – the newly added instance

Return type *PhotonicInstance*

add_monitor_obj(*monitor_obj*)
Add a new Lumerical monitor object to the db

add_obj(*obj*)
Takes a provided layout object and adds it to the db. Automatically detects what type of object is being added, and sends it to the appropriate category in the layoutDB. Also accepts a list of layout objects.

TODO: Provide support for directly adding photonic ports and simulation objects

add_path(*layer: Union[str, Tuple[str, str]], width: Union[float, int], points: List[Tuple[Union[float, int], Union[float, int]]], resolution: float, end_style: str = 'truncate', join_style: str = 'extend', unit_mode: bool = False*) → BPG.objects.PhotonicPath
Creates a PhotonicPath object based on the provided arguments and adds it to the db

add_photonic_port(*name: str = None, center: Tuple[Union[float, int], Union[float, int]] = None, orient: str = None, width: Union[float, int] = None, layer: Union[str, Tuple[str, str]] = None, overwrite_purpose: bool = False, resolution: float = None, unit_mode: bool = False, port: BPG.port.PhotonicPort = None, overwrite: bool = False, show: bool = True*) → BPG.port.PhotonicPort
Adds a photonic port to the current hierarchy. A PhotonicPort object can be passed, or will be constructed if the proper arguments are passed to this function.

Parameters

- **name** (*str*) – name to give the new port
- **center** (*coord_type*) – (x, y) location of the port
- **orient** (*str*) – orientation pointing INTO the port
- **width** (*dim_type*) – the port width
- **layer** (*Union[str, Tuple[str, str]]*) – the layer on which the port should be added. If only a string, the purpose is defaulted to 'port'
- **overwrite_purpose** (*bool*) – True to overwrite the 'port' purpose if an LPP is passed. If False (default), the purpose of a passed LPP is stripped away and the 'port' purpose is used.
- **resolution** (*Union[float, int]*) – the grid resolution
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units
- **port** (*Optional[PhotonicPort]*) – the PhotonicPort object to add. This argument can be provided in lieu of all the others.
- **overwrite** (*bool*) – True to add the port with the specified name even if another port with that name already exists in this level of the design hierarchy.
- **show** (*bool*) – True to draw the port indicator shape

Returns **port** – the added photonic port object

Return type *PhotonicPort*

add_polygon(*layer: Union[str, Tuple[str, str]] = None, points: List[Tuple[Union[float, int], Union[float, int]]] = None, resolution: float = None, unit_mode: bool = False*) → BPG.objects.PhotonicPolygon
Creates a new polygon from the user provided points and adds it to the db

Parameters

- **layer** (*Union[str, Tuple[str, str]]*) – the layer of the polygon

- **resolution** (*float*) – the layout grid resolution
- **points** (*List[coord_type]*) – the points defining the polygon
- **unit_mode** (*bool*) – True if the points are given in resolution units

Returns **polygon** – the added polygon object

Return type *PhotonicPolygon*

add_rect(*layer: Union[str, Tuple[str, str]], coord1: Tuple[Union[float, int], Union[float, int]] = None, coord2: Tuple[Union[float, int], Union[float, int]] = None, bbox: Union[bag.layout.util.BBox, bag.layout.util.BBoxArray] = None, nx: int = 1, ny: int = 1, spx: Union[float, int] = 0, spy: Union[float, int] = 0, unit_mode: bool = False*) → BPG.objects.PhotonicRect

Creates a new rectangle based on the user provided arguments and adds it to the db. User can either provide a pair of coordinates representing opposite corners of the rectangle, or a BBox/BBoxArray. This rectangle can also be arrayed with the number and spacing parameters.

Parameters

- **layer** (*Union[str, Tuple[str, str]]*) – the layer name, or the (layer, purpose) pair.
- **coord1** (*Tuple[Union[int, float], Union[int, float]]*) – point defining one corner of rectangle boundary.
- **coord2** (*Tuple[Union[int, float], Union[int, float]]*) – opposite corner from coord1 defining rectangle boundary.
- **bbox** (*bag.layout.util.BBox or bag.layout.util.BBoxArray*) – the base bounding box. If this is a BBoxArray, the BBoxArray's arraying parameters are used.
- **nx** (*int*) – number of columns.
- **ny** (*int*) – number of rows.
- **spx** (*float*) – column pitch.
- **spy** (*float*) – row pitch.
- **unit_mode** (*bool*) – True if layout dimensions are specified in resolution units.

Returns **rect** – the added rectangle.

Return type *PhotonicRect*

add_round(*layer: Union[str, Tuple[str, str]], resolution: float, rout: Union[float, int], center: Tuple[Union[float, int], Union[float, int]] = (0, 0), rin: Union[float, int] = 0, theta0: Union[float, int] = 0, theta1: Union[float, int] = 360, nx: int = 1, ny: int = 1, spx: Union[float, int] = 0, spy: Union[float, int] = 0, unit_mode: bool = False*)

Creates a PhotonicRound object based on the provided arguments and adds it to the db

add_sim_obj(*sim_obj*)

Add a new Lumerical simulation object to the db

add_source_obj(*source_obj*)

Add a new Lumerical source object to the db

add_via_stack(*bot_layer: Union[str, Tuple[str, str]], top_layer: Union[str, Tuple[str, str]], loc: Tuple[Union[float, int], Union[float, int]], min_area_on_bot_top_layer: bool = False, unit_mode: bool = False*)

Adds a via stack with one via in each layer at the provided location. All intermediate layers will be enclosed with an enclosure that satisfies both via rules and min area rules

Parameters

- **bot_layer** (*Union[str, Tuple[str, str]]*) – Layer name or layer LPP of the bottom layer in the via stack
- **top_layer** (*Union[str, Tuple[str, str]]*) – Layer name or layer LPP of the top layer in the via stack
- **loc** (*coord_type*) – Coordinate of the center of the via stack
- **min_area_on_bot_top_layer** (*bool*) – True to have enclosures on top and bottom layer satisfy minimum area constraints
- **unit_mode** (*bool*) – True if input arguments are specified in layout resolution units

add_via_stack_by_ind(*bot_layer_ind: int, top_layer_ind: int, loc: Tuple[Union[float, int], Union[float, int]], min_area_on_bot_top_layer: bool = False, unit_mode: bool = False*)

Adds a stack of vias from the metal at the bot_layer_ind index to the metal at the top_layer_ind index.

Parameters

- **bot_layer_ind** (*int*) – Index of the bottom layer of the via stack
- **top_layer_ind** (*int*) – Index of the top layer of the via stack
- **loc** (*coord_type*) – Coordinate of the center of the via stack
- **min_area_on_bot_top_layer** (*bool*) – True to have enclosures on top and bottom layer satisfy minimum area constraints
- **unit_mode** (*bool*) – True if input arguments are specified in layout resolution units

delete_port(*port_names: Union[str, List[str]]*) → None

Removes the given ports from this instances list of ports. Raises error if given port does not exist.

Parameters **port_names** (*Union[str, List[str]]*) –

draw_layout()

Draw the layout of this template.

Override this method to create the layout.

WARNING: you should never call this method yourself.

extract_photonic_ports(*inst: Union[BPG.objects.PhotonicInstance, Instance], port_names: Union[str, List[str], None] = None, port_renaming: Optional[Dict[str, str]] = None, unmatched_only: bool = True, show: bool = True*) → None

Brings ports from lower level of hierarchy to the current hierarchy level

Parameters

- **inst** (PhotonicInstance) – the instance that contains the ports to be extracted
- **port_names** (*Optional[Union[str, List[str]]*) – the port name or list of port names re-export. If not supplied, all ports of the inst will be extracted
- **port_renaming** (*Optional[Dict[str, str]]*) – a dictionary containing key-value pairs mapping inst's port names (key) to the new desired port names (value). If not supplied, extracted ports will be given their original names
- **unmatched_only** (*bool*) –
- **show** (*bool*) –

finalize()
Finalize this master instance.

get_photonic_port(*port_name: Optional[str] = ''*) → BPG.port.PhotonicPort
Returns the photonic port object with the given name

Parameters **port_name** (*Optional[str]*) – the photonic port terminal name. If None or empty, check if this photonic template has only one port, and return it

Returns **port** – The photonic port object

Return type *PhotonicPort*

has_photonic_port(*port_name: str*) → bool
Checks if the given port name exists in the current hierarchy level.

Parameters **port_name** (*str*) – the name of the port

Returns

- *boolean*
- *true if port exists in current hierarchy level*

photonic_ports_names_iter() → Iterable[str]

update_port()

5.1.14 Module contents

BPG.**check_environment**()
Checks that all required environment variables have been set

BPG.**setup_environment**()
Sets up python module search path from config file

PYTHON MODULE INDEX

b

INDEX

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z