picwriter Documentation

Release 0.0.1

Derek Kita

Jul 24, 2019

Contents

Picwriter (Photonic-Integrated-Circuit Writer) is a free Python module, built above the gdspy module, aimed at simplifying the process of designing complex GDSII masks for photonic integrated circuits through a prebuilt library of easy-to-implement PCells. Supported blocks include waveguides, straight grating couplers, focusing grating couplers, tapers, directional couplers, multi-mode interferometers (MMI's), resonators, spiral structures, and more that are coming soon!

CHAPTER 1

Features

The ultimate goal of this module is to reduce the time required to generate photonic integrated circuit mask designs, by extending the functionality of the gdspy library.

- High-level specification of common building blocks for photonic-integrated circuits
- Easily snap photonic components together using portlist syntax and waypoint routing of waveguides and metal traces
- PICwriter will automatically detect if you are adding a cell to the mask which is identical to one added before, so you don't have to worry about referencing existing cells.
- Waveguide bends and curves automatically compute the number of vertices per polygon to minimize grid errors.
- (coming soon!) simple templates for writing your own custom PCells to be used with PICwriter

CHAPTER 2

Contribute

This project got started because I wanted to make it easier to generate simple lithography masks for in-house fabrication of PICs at MIT. If you find this project useful, or have ideas for new components to add to the library, please feel free to contribute to the project!

- Issue Tracker: github.com/DerekK88/PICwriter/issues
- Source Code: github.com/DerekK88/PICwriter

2.1 Guide

2.1.1 Installation

PICwriter is tested on python versions 2.7, 3.4, 3.5, and 3.6 on Linux, OS X, and Windows. Please check here for the current build status (if building from source).

Requirements

A working version of python is required for using the PICwriter library. You can go to python.org to download python (or check if it's installed on your computer by running *python –version* in a command prompt or terminal. I personally recommend downloading Anaconda since it includes several nice scientific libraries, the conda package manager, Spyder IDE, and other niceties.

Installation (Linux / OS X)

(**Option 1 – preferred**) Install PICwriter by first downloading the source code here. and then in the picwriter directory run:

```
python setup.py install
```

(**Option 2:**) Install PICwriter by running:

```
pip install picwriter
```

Installation (Windows)

The best way of obtaining the library is by installing the prebuilt binaries.

- First, go to the gdspy appveyor project page, then click the python environment that matches your python version and processor type. For example, if you have a 64-bit processor with Python version 3.5 (you can check by running *python –version* in a command prompt) then you would click 'PYTHON=C:Python35-x64'. Then, click the **Artifacts** tab and download the corresponding *distgdspy-1.X.X.X.whl* wheel file.
- Open up a command prompt (type *cmd* in the search bar), navigate to your downloads, then install the appropriate *.whl* file via:

```
pip install gdspy-1.X.X.X.whl
```

- Next, install the PICwriter library by following the same procedure as before at the picwriter appveyor page to install the corresponding prebuilt picwriter *.whl* file.
- In a command prompt, navigate to your downloads and install the appropriate *.whl* file with pip:

```
pip install picwriter-1.X.X.X.whl
```

Building from source is also possible. For installing gdspy, an appropriate build environment is required for compilation of the C extension modules.

Simulations with PICwriter

In order to use the built-in PICwriter simulation functions, it is necessary to download and build MEEP and MPB (the free and open-source software packages for electromagnetic simulation that PICwriter calls). Currently, this functionality is only tested on Ubuntu 16.04, but should work fine for any UNIX environment. For the FDTD functionality, it is necessary for MEEP and MPB to be built from source (so that they can be linked together for importing eigenmode sources). Below is a downloadable script that I use to install and build MEEP/MPB from source: `build_meep_python_parallel.sh`.

Please see the section *Simulations with PICwriter* for information and tutorials for simulating PICwriter components.

Getting Started

You can check that PICwriter and gdspy are properly installed by running:

```
import gdspy
import picwriter
```

in a python file or python prompt. To get a feel for using the PICwriter library, checkout the Tutorial page.

2.1.2 Tutorial

Getting started with PICwriter is easy. In the examples below, we'll walk you through quickly generating a couple lithography masks.

Basic code structure

Every file should start with the following import statements:

```
import gdspy
from picwriter import toolkit as tk
import picwriter.components as pc
```

The first statement allows us to use the base commands from the gdspy library. The second provides additional functionality for working with the picwriter components, which are imported with the third statement. Next, we can create a top-level cell that we will add all of our PIC components to:

```
top = gdspy.Cell("top")
```

Note: some IDE's such as Spyder do not reload the GdsLibrary() between subsequent runs, and so adding *gdspy.current_library = gdspy.GdsLibrary()* below the import statements may fix potential name-clashes. Now, we can just add a simple geometric shape from the gdspy library, such as a square:

```
top.add(gdspy.Rectangle((0,0), (1000, 1000), layer=100, datatype=0))
```

Let's add a simple waveguide with a bend. To do this, we only need a reference to a WaveguideTemplate reference, and a set of waypoints (i.e. a list of (x,y) tuples). The WaveguideTemplate class specifies all important parameters, such as resist type, waveguide width, cladding width, bending radius, then the layer and datatype:

```
wgt = pc.WaveguideTemplate(wg_width=0.45, clad_width=10.0, bend_radius=100, resist='+
↪', fab='ETCH',
                           wg_layer=1, wg_datatype=0, clad_layer=2, clad_datatype=0)
wg = pc.Waveguide([(25, 25), (975, 25), (975,500), (25,500),(25,975),(975,975)], wgt)
```

We then add this to the top cell using the toolkit "add" method:

```
tk.add(top, wg)
```

This is simply a shortcut for the gdspy add method (which would look like *top.add(gdspy.CellReference(subcell))*. Both are equivalent, though the first is slightly less typing. To generate the mask according to the fab specifications (positive/negative resist, and fabrication type), we can call the *tk.build_mask()* function:

```
tk.build_mask(top, wgt, final_layer=3, final_datatype=0)
```

This simply takes the waveguide layer and the cladding layer, then does the appropriate 'xor' operation (or just simply returns the waveguide layer). Alternatively, you could just skip this and use KLayout (or an equivalent application) to perform the necessary boolean operations. Lastly, we can visualize everything by either visualizing with the built-in gdspy LayoutViewer, or exporting to a GDSII file in the working directory of your python script:

```
gdspy.LayoutViewer()
gdspy.write_gds('tutorial.gds', unit=1.0e-6, precision=1.0e-9)
```

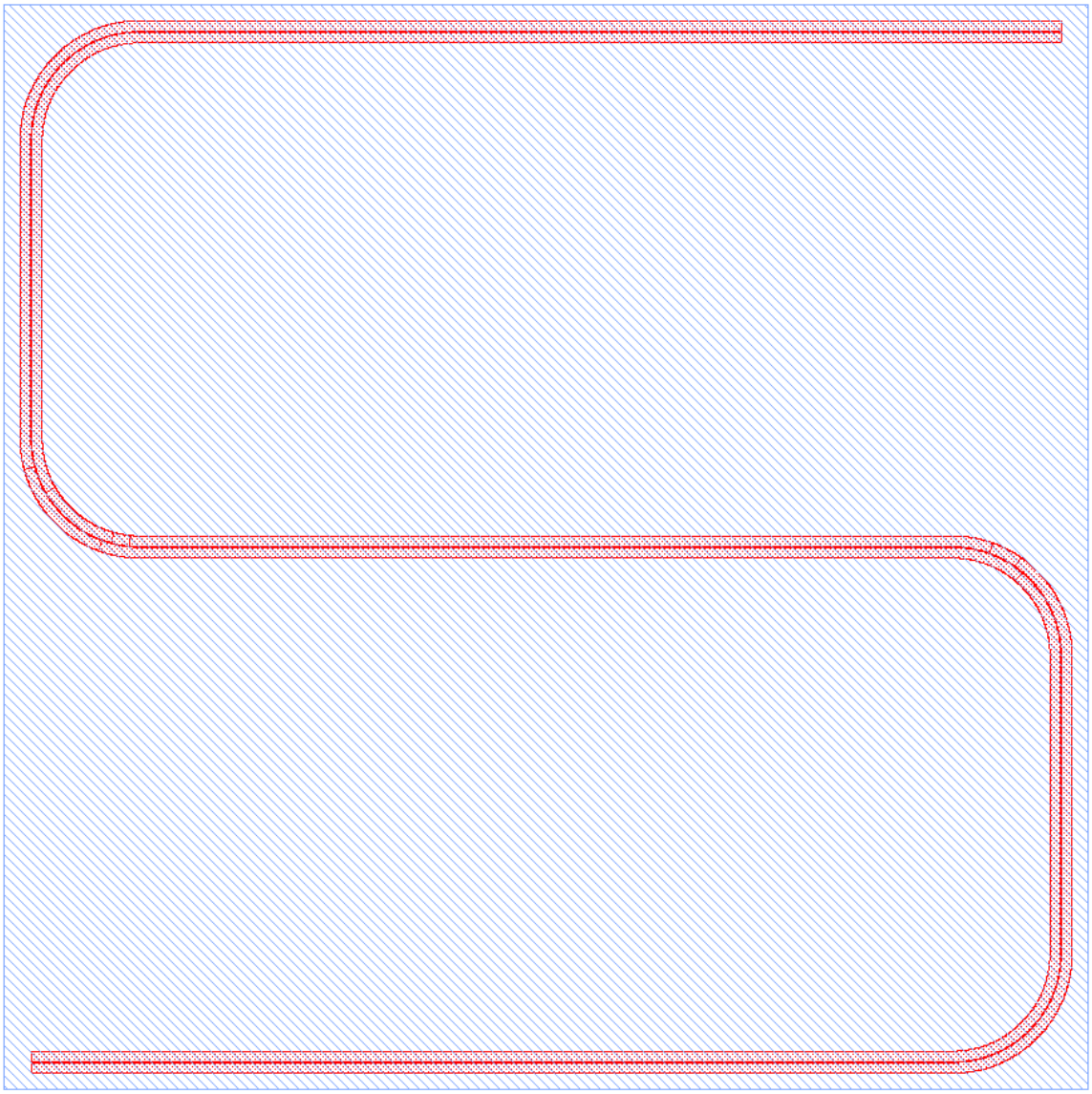
The 'units' specifies we are using microns as the base unit, and 'precision' specifies 1 nm precision.

Tutorial 1 program

Below is the entire program, along with an image of the GDSII file it generates:

```
import gdspy
from picwriter import toolkit as tk
import picwriter.components as pc

top = gdspy.Cell("top")
wgt = pc.WaveguideTemplate(wg_width=0.45, clad_width=10.0, bend_radius=100, resist='+
↪', fab='ETCH',
                           wg_layer=1, wg_datatype=0, clad_layer=2, clad_datatype=0)

top.add(gdspy.Rectangle((0,0), (1000, 1000), layer=100, datatype=0))
wg = pc.Waveguide([(25, 25), (975, 25), (975,500), (25,500),(25,975),(975,975)], wgt)
tk.add(top, wg)

tk.build_mask(top, wgt, final_layer=3, final_datatype=0)

gdspy.LayoutViewer()
gdspy.write_gds('tutorial.gds', unit=1.0e-6, precision=1.0e-9)
```

The results should look like this, if we select only the final layer (3/0) and layer for the square (100/0):



Putting multiple components together

Here we show how to generate a slightly more complex mask using the set of supported components that come standard in the picwriter library. Let's build up an interesting object, a Mach-Zehnder interferometer, which consists of two 1x2 MMI's, and one arm that is much longer than the other. But to make it longer, we need to use a *spiral* type of waveguide (for compactness).

Unbalanced Mach-Zehnder interferometer with spiral arm

As usual, we begin with the import statements, a 'top' cell, and a standard waveguide template. We then place a grating coupler for light to be coupled onto the PIC. The 'port' and 'direction' keyword arguments (kwargs) specify the location of the port, and the direction that light goes *into* the device, so we specify (0,0) and *'WEST'*. We also specify a waveguide that extends for 200 um:

```
import gdspy
from picwriter import toolkit as tk
import picwriter.components as pc

top = gdspy.Cell('top')
wgt = pc.WaveguideTemplate(wg_width=0.45, clad_width=10.0, bend_radius=60, resist='+',
↪ fab='ETCH', wg_layer=1, wg_datatype=0, clad_layer=2, clad_datatype=0)

gc1 = pc.GratingCouplerFocusing(wgt, focus_distance=20.0, width=20, length=40,␣
↪period=1.0, dutycycle=0.7, port=(100,0), direction='WEST')
tk.add(top, gc1)

wg1 = pc.Waveguide([gc1.portlist['output']['port'], (200,0)], wgt)
tk.add(top, wg1)
```

We can now add a 1x2 MMI. Rather than calculate the location of each port and direction the waveguide should go in, we can *unpack* the relevant port and direction information from the component we are connecting to. We do this by passing ***wg1.portlist['output']* to the input of the MMI1x2 class. wg1.portlist is simply a python dictionary that contains the keys 'port' and 'direction', and the two asterisks unpack the corresponding port and direction values the new MMI1x2 object. We can then add the second MMI, which will be some distance away from the first one:

```
mmi1 = pc.MMI1x2(wgt, length=50, width=10, taper_width=2.0, wg_sep=3, **wg1.portlist[
↪'output'])
tk.add(top, mmi1)

mmi2 = pc.MMI1x2(wgt, length=50, width=10, taper_width=2.0, wg_sep=3, port=(1750, 0),␣
↪direction='WEST')
tk.add(top, mmi2)
```

We can explicitly get the (x,y) value of the ports by referencing the corresponding 'port' in the MMI's portlist, then use these values to build a waveguide bend up towards where we will create the spiral:

```
(xtop, ytop) = mmi1.portlist['output_top']['port']
wg2 = pc.Waveguide([(xtop, ytop),
                    (xtop+100, ytop),
                    (xtop+100, ytop+200),
                    (xtop+200, ytop+200)], wgt)
tk.add(top, wg2)
```

Next, we add the spiral at the location where the previous waveguide ended. Then at the output of the spiral, we place another waveguide connecting to the spiral output to the 2x1 MMI:

```
sp = pc.Spiral(wgt, 600.0, 1000.0, 8000.0, parity=-1, **wg2.portlist['output'])
tk.add(top, sp)

(xtop_out, ytop_out) = sp.portlist['output']['port']
(xmmi_top, ymmi_top) = mmi2.portlist['output_bot']['port']
wg_spiral_out = Waveguide([(xtop_out, ytop_out),
                           (xmmi_top-100, ytop_out),
                           (xmmi_top-100, ytop_out-200),
                           (xmmi_top, ytop_out-200)], wgt)
```

We then add a waveguide for the bottom 'arm' of the Mach-Zehnder that directly connects the first MMI to the second MMI:

```
(xbot, ybot) = mmi1.portlist['output_bot']['port']
wg3 = pc.Waveguide([(xbot, ybot),
```

(continues on next page)

(continued from previous page)

```
              (xbot+100, ybot),
              (xbot+100, ybot-200),
              (xmmi_top-100, ybot-200),
              (xmmi_top-100, ybot),
              (xmmi_top, ybot)], wgt)
tk.add(top, wg3)
```

The last grating coupler then is placed at the location of the second MMI 'port', plus an additional 100 um in the '+x' direction:

```
gc2 = pc.GratingCouplerFocusing(wgt, focus_distance=20.0, width=20, length=40,␣
↪period=1.0, dutycycle=0.7,
port=(mmi2.portlist['input']['port'][0]+100, mmi2.portlist['input']['port'][1]),␣
↪direction='EAST')
tk.add(top, gc2)

wg_gc2 = pc.Waveguide([mmi2.portlist['input']['port'], gc2.portlist['output']['port
↪']], wgt)
tk.add(top, wg_gc2)
```

Our mask is now ready to be 'built' and visualized:

```
tk.build_mask(top, wgt, final_layer=3, final_datatype=0)

gdspy.LayoutViewer()
gdspy.write_gds('tutorial2.gds', unit=1.0e-6, precision=1.0e-9)
```

Tutorial 2 program

Altogether, the entire code for the example is shown below:

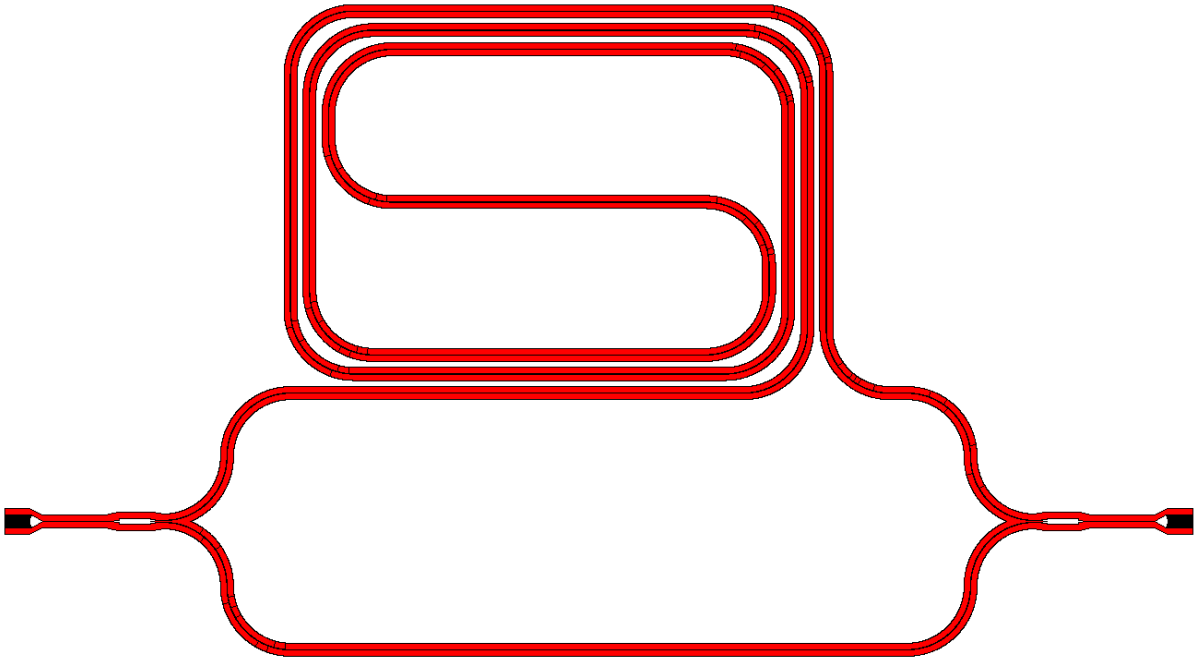
```
import gdspy
from picwriter import toolkit as tk
import picwriter.components as pc

top = gdspy.Cell('top')
wgt = pc.WaveguideTemplate(wg_width=0.45, clad_width=10.0, bend_radius=60, resist='+',
↪ fab='ETCH', wg_layer=1, wg_datatype=0, clad_layer=2, clad_datatype=0)

gc1 = pc.GratingCouplerFocusing(wgt, focus_distance=20.0, width=20, length=40,␣
↪period=1.0, dutycycle=0.7, port=(100,0), direction='WEST')
tk.add(top, gc1)

wg1 = pc.Waveguide([gc1.portlist['output']['port'], (200,0)], wgt)
tk.add(top, wg1)

mmi1 = pc.MMI1x2(wgt, length=50, width=10, taper_width=2.0, wg_sep=3, **wg1.portlist[
↪'output'])
tk.add(top, mmi1)

mmi2 = pc.MMI1x2(wgt, length=50, width=10, taper_width=2.0, wg_sep=3, port=(1750, 0),␣
↪direction='WEST')
tk.add(top, mmi2)

(xtop, ytop) = mmi1.portlist['output_top']['port']
```

(continues on next page)

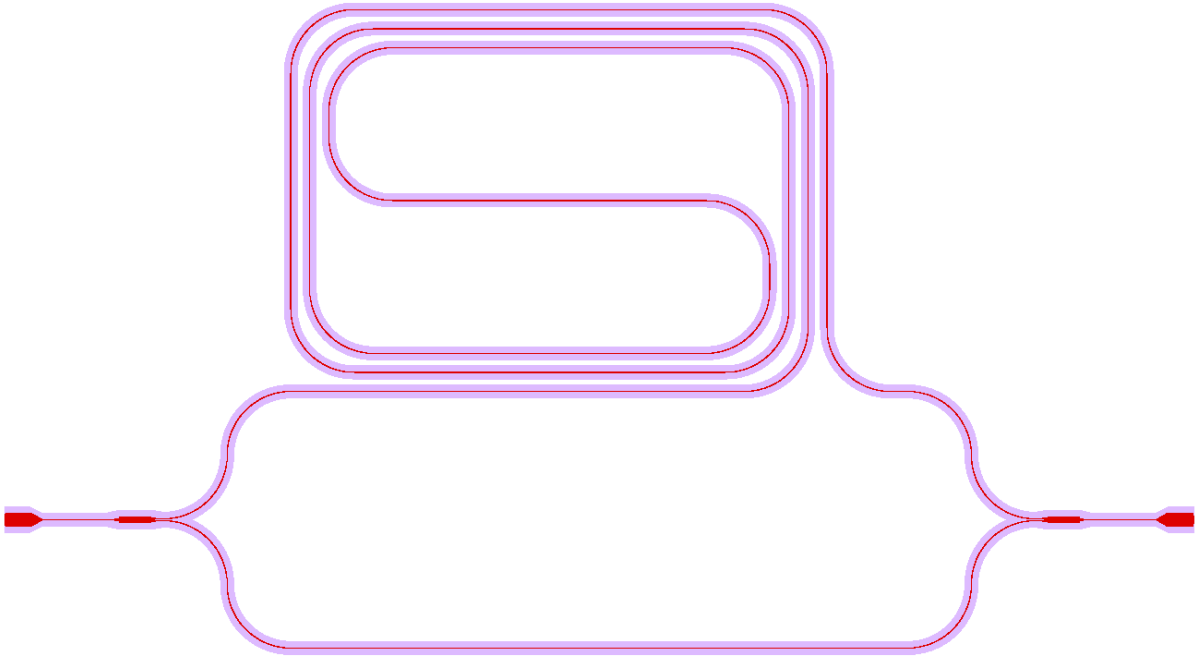
(continued from previous page)

```
wg2 = pc.Waveguide([(xtop, ytop),
              (xtop+100, ytop),
              (xtop+100, ytop+200),
              (xtop+200, ytop+200)], wgt)
tk.add(top, wg2)

sp = pc.Spiral(wgt, 800.0, 8000.0, parity=-1, **wg2.portlist['output'])
tk.add(top, sp)

(xtop_out, ytop_out) = sp.portlist['output']['port']
(xmmi_top, ymmi_top) = mmi2.portlist['output_bot']['port']
wg_spiral_out = pc.Waveguide([(xtop_out, ytop_out),
                         (xmmi_top-100, ytop_out),
                         (xmmi_top-100, ytop_out-200),
                         (xmmi_top, ytop_out-200)], wgt)
tk.add(top, wg_spiral_out)

(xbot, ybot) = mmi1.portlist['output_bot']['port']
wg3 = pc.Waveguide([(xbot, ybot),
              (xbot+100, ybot),
              (xbot+100, ybot-200),
              (xmmi_top-100, ybot-200),
              (xmmi_top-100, ybot),
              (xmmi_top, ybot)], wgt)
tk.add(top, wg3)

gc2 = pc.GratingCouplerFocusing(wgt, focus_distance=20.0, width=20, length=40,␣
↪period=1.0, dutycycle=0.7,
port=(mmi2.portlist['input']['port'][0]+100, mmi2.portlist['input']['port'][1]),␣
↪direction='EAST')
tk.add(top, gc2)

wg_gc2 = pc.Waveguide([mmi2.portlist['input']['port'], gc2.portlist['output']['port
↪']], wgt)
tk.add(top, wg_gc2)

tk.build_mask(top, wgt, final_layer=3, final_datatype=0)

gdspy.LayoutViewer(cells=top)
gdspy.write_gds('tutorial2.gds', unit=1.0e-6, precision=1.0e-9)
```

The resulting GDSII file looks like this:



And the waveguide & cladding layers that are generated are shown below:



Generating Hierarchical PCells

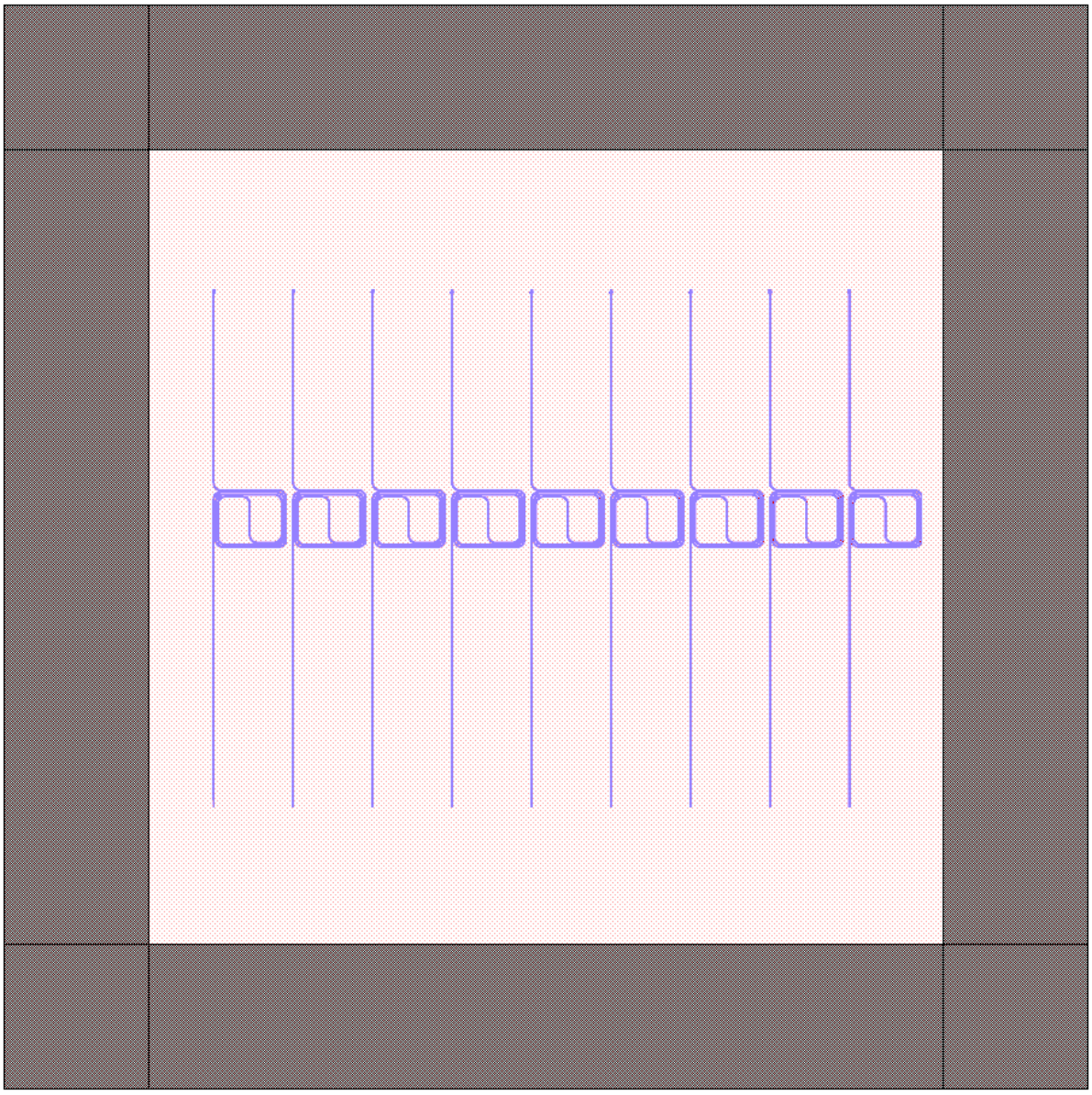
In the next example, we show how easy it is to generate and reuse hierarchical PCells to quickly populate a mask with many similar components in different locations. In the program below, we create a gdspy.Cell class called 'spiral_unit', then add different components, just like we would have added them to the 'top' cell before. However, we can now create multiple references to this cell through gdspy.CellReference, and place them in several different locations on our mask. This has several advantages: (1) the time to making a mask with repeating units is reduced, and (2) the GDSII file size is reduced since we only need to store the information about one 'cell' and the locations of all the cell references (as opposed to storing the memory for each cell multiplied by the number of cells we place!). The full program is below:

```
import numpy as np
import gdspy
from picwriter import toolkit as tk
import picwriter.components as pc

X_SIZE, Y_SIZE = 15000, 15000
exclusion_region = 2000.0 #region where no devices are to be fabricated
x0, y0 = X_SIZE/2.0, Y_SIZE/2.0 #define origin of the die
step = 100.0 #standard spacing between components

top = gdspy.Cell("top")

wgt = pc.WaveguideTemplate(wg_width=0.45, clad_width=10.0, bend_radius=100,
                      resist='+', fab='ETCH', wg_layer=1, wg_datatype=0,
                      clad_layer=2, clad_datatype=0)

""" Add a die outline, with exclusion, from gdspy geometries found at
http://gdspy.readthedocs.io/en/latest/"""
top.add(gdspy.Rectangle((0,0), (X_SIZE, Y_SIZE), layer=6, datatype=0))
top.add(gdspy.Rectangle((0, Y_SIZE-exclusion_region), (X_SIZE, Y_SIZE), layer=7,␣
↪datatype=0))
top.add(gdspy.Rectangle((0, 0), (X_SIZE, exclusion_region), layer=7, datatype=0))
top.add(gdspy.Rectangle((0, 0), (exclusion_region, Y_SIZE), layer=7, datatype=0))
top.add(gdspy.Rectangle((X_SIZE-exclusion_region, 0), (X_SIZE, Y_SIZE), layer=7,␣
↪datatype=0))

""" Add some components from the PICwriter library """
spiral_unit = gdspy.Cell("spiral_unit")
sp1 = pc.Spiral(wgt, 1000.0, 10000, parity=1, port=(500.0+exclusion_region+4*step,y0))
tk.add(spiral_unit, sp1)

wg1=pc.Waveguide([sp1.portlist["input"]["port"], (sp1.portlist["input"]["port"][0],␣
↪4000.0)], wgt)
wg2=pc.Waveguide([sp1.portlist["output"]["port"], (sp1.portlist["output"]["port"][0],␣
↪Y_SIZE-4000.0)], wgt)
tk.add(spiral_unit, wg1)
tk.add(spiral_unit, wg2)

tp_bot = pc.Taper(wgt, length=100.0, end_width=0.1, **wg1.portlist["output"])
tk.add(spiral_unit, tp_bot)

gc_top = pc.GratingCouplerFocusing(wgt, focus_distance=20, width=20, length=40,
                              period=0.7, dutycycle=0.4, wavelength=1.55,
                              sin_theta=np.sin(np.pi*8/180), **wg2.portlist["output"])
tk.add(spiral_unit, gc_top)
```

(continues on next page)

(continued from previous page)

```
for i in range(9):
   top.add(gdspy.CellReference(spiral_unit, (i*1100.0, 0)))

tk.build_mask(top, wgt, final_layer=3, final_datatype=0)

gdspy.LayoutViewer(cells=top)
gdspy.write_gds('mask_template.gds', unit=1.0e-6, precision=1.0e-9)
```



2.1.3 Component Documentation

Below is the set of supported components that come standard in the picwriter library:

Waveguides & Waveguide Templates

Waveguide Template

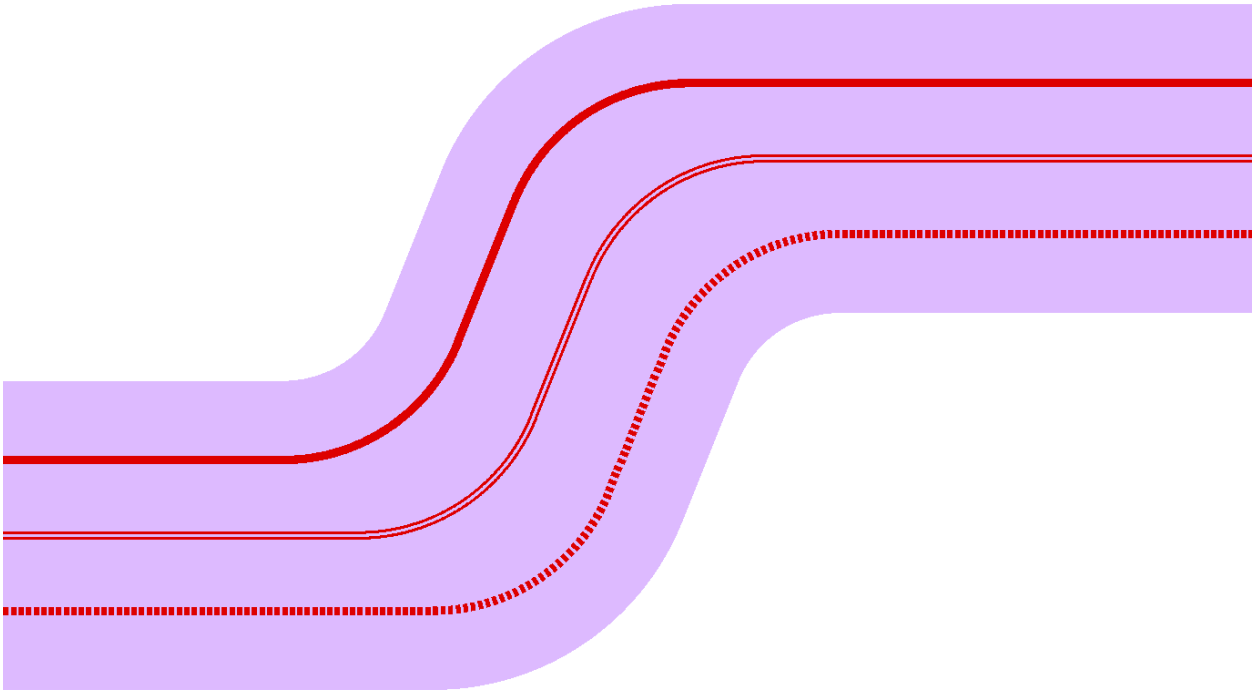
Waveguide template objects are used to define a standard set of parameters (width, cladding, layers, etc.) that is passed to waveguide routes and PICwriter components.

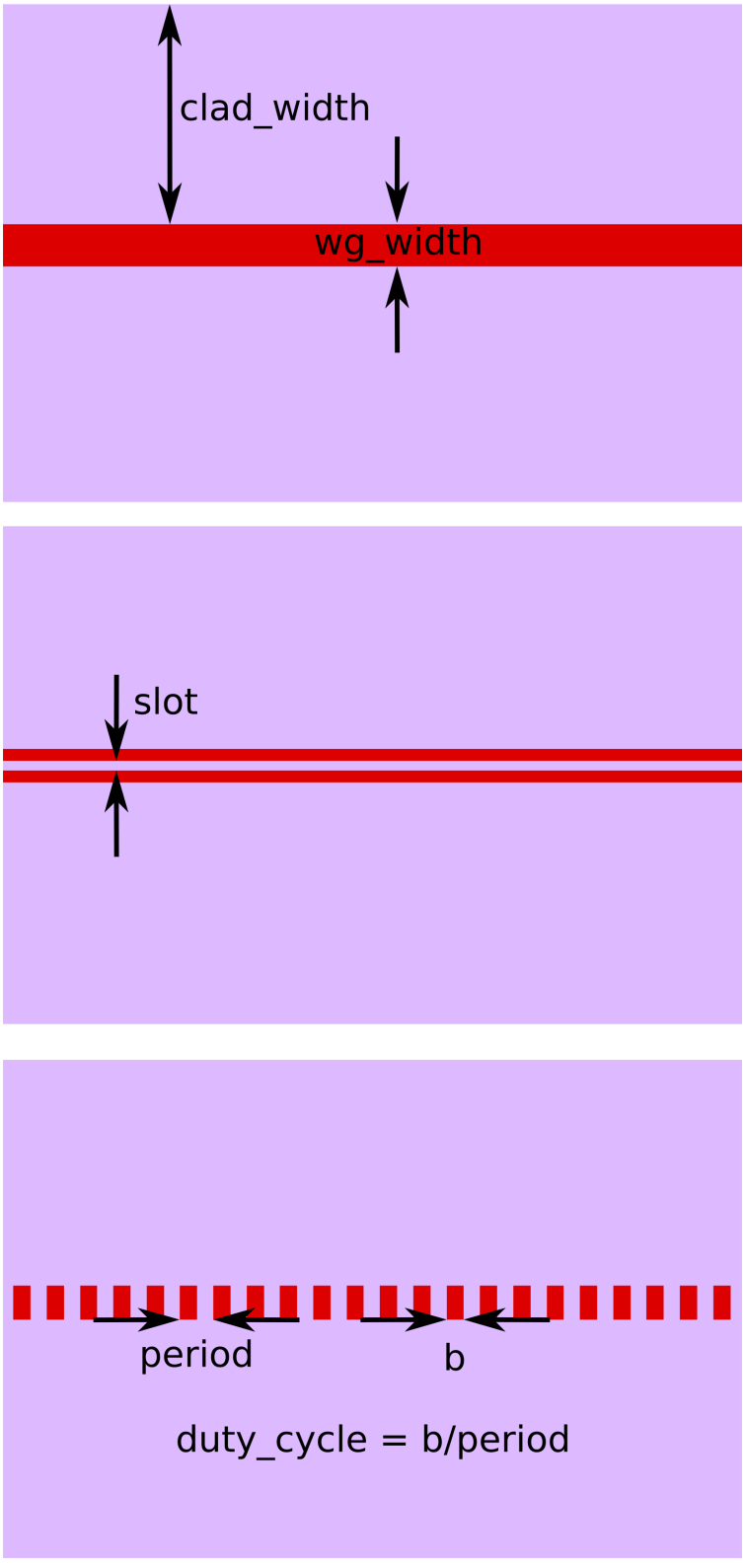
class picwriter.components.**WaveguideTemplate**(*wg_type=u'strip'*, *bend_radius=50.0*, *waveguide_stack=None*, *wg_width=2.0*, *clad_width=10.0*, *grid=0.001*, *resist=u'+'*, *fab=u'ETCH'*, *slot=0.1*, *period=0.1*, *duty_cycle=0.5*, *wg_layer=1*, *wg_datatype=0*, *clad_layer=2*, *clad_datatype=0*, *euler_bend=False*)

Template for waveguides that contains standard information about the geometry and fabrication. Supported waveguide types are **strip** (also known as 'channel' waveguides), **slot**, and **SWG** ('sub-wavelength grating', or 1D photonic crystal waveguides).

Keyword Args:

- **wg_type** (string): Type of waveguide used. Options are 'strip', 'slot', and 'swg'. Defaults to 'strip'.
- **bend_radius** (float): Radius of curvature for waveguide bends (circular). Defaults to 50.
- **waveguide_stack** (list): List of layers and path widths to be drawn when waveguides are routed & placed. Format is '[[width1, (layer1, datatype1)], [width2, (layer2, datatype2)], ...]'. The first element defines the main waveguide width & layer for slot and subwavelength gratings. If using waveguide_stack, the following keyword arguments are ignored: wg_width, clad_width, wg_layer, wg_datatype, clad_layer, clad_datatype. Defaults to [[2.0, (1,0)], [10.0, (2,0)]].
- **wg_width** (float): Width of the waveguide as shown on the mask. Defaults to 2.
- **euler_bend** (boolean): If *True*, uses Euler bends to route waveguides. Defaults to *False*. Currently only works with slot and strip waveguides. The given *bend_radius* value determines the **smallest** bend radius along the entire Euler curve.
- **slot** (float): Size of the waveguide slot region. This is only used if *wg_type*'='slot'‘. Defaults to 0.1.
- **period** (float): Period of the SWG. This is only used if *wg_type*'='swg'‘. Defaults to 0.1.
- **duty_cycle** (float): Duty cycle of the SWG. This is only used if *wg_type*'='swg'‘. Defaults to 0.5.
- **clad_width** (float): Width of the cladding (region next to waveguide, mainly used for positive-type photoresists + etching, or negative-type and liftoff). Defaults to 10.
- **grid** (float): Defines the grid spacing in units of microns, so that the number of points per bend can be automatically calculated. Defaults to 0.001 (1 nm).
- **resist** (string): Must be either '+' or '-'. Specifies the type of photoresist used. Defaults to '+'
- **fab** (string): If 'ETCH', then keeps resist as is, otherwise changes it from '+' to '-' (or vice versa). This is mainly used to reverse the type of mask used if the fabrication type is 'LIFTOFF'. Defaults to 'ETCH'.
- **wg_layer** (int): Layer type used for waveguides. Defaults to 1.
- **wg_datatype** (int): Data type used for waveguides. Defaults to 0.
- **clad_layer** (int): Layer type used for cladding. Defaults to 2.
- **clad_datatype** (int): Data type used for cladding. Defaults to 0.





Example usage:

To generate a strip waveguide template with 1.0um width and 25.0um bending radius:

```
wgt= WaveguideTemplate(wg_type='strip', wg_width=1.0, bend_radius=25)
```

To generate a slot waveguide with 1.0um width, 25.0um bending radius, and a 0.3um slot in the center:

```
wgt = WaveguideTemplate(wg_type='slot', wg_width=1.0, bend_radius=25.0, slot=0.3)
```

To generate a sub-wavelength grating waveguide with 1.0um width, 25.0um bending radius, 50% duty-cycle, and a 1um period:

```
wgt= WaveguideTemplate(wg_type='swg', wg_width=1.0, bend_radius=25, duty_cycle=0.50,␣
↪period=1.0)
```

Waveguides

Waveguide objects are fully defined by a WaveguideTemplate object as well as a list of (x,y) points that determine where the waveguide is routed.

class picwriter.components.**Waveguide**(*trace*, *wgt*)

Waveguide Cell class.

Args:

- **trace** (list): List of coordinates used to generate the waveguide (such as '[(x1,y1), (x2,y2), . . .]').
- **wgt** (WaveguideTemplate): WaveguideTemplate object

Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) are the first elements of 'trace', (x2, y2) are the last elements of 'trace', and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the component that the waveguide will connect to.

Example usage to generate a waveguide with waypoints:

```
top = gdspy.Cell('top')
wgt= WaveguideTemplate(wg_type='strip', wg_width=1.0, bend_radius=25)
waypoints = [(0, 0), (200, 0), (250, 100), (400, 100)]
wg = Waveguide(waypoints, wgt)
tk.add(top, wg)
gdspy.LayoutViewer()
```

Bends

Euler Bends

class picwriter.components.**EBend**(*wgt*, *turnby*, *port=(0, 0)*, *direction=u'EAST'*, *vertex=None*)

Euler shaped Bend Cell class. Creates a generic Euler waveguide bend that can be used in waveguide routing. The number of points is computed based on the waveguide template grid resolution to automatically minimize grid errors. This class can be automatically called and implemented during waveguide routing by passing

euler_bend=True to a WaveguideTemplate object. The smallest radius of curvature on the Euler bend is set to be the *bend_radius* value given by the WaveguideTemplate object passed to this class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object. Bend radius is extracted from this object.
- **turnby** (float): Angle in radians, must be between +np.pi and -np.pi. It's not recommended that you give a value of np.pi (for 180 bends) as this will result in divergent trig identities. Instead, use two bends with turnby=Pi/2.

Keyword Args:

- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)
- **vertex** (tuple): If a value for *vertex* is given (Cartesian x,y coordinate), then the Euler bend is placed at this location, bypassing the normal *port* value. This is used in waypoint routing.

Members:

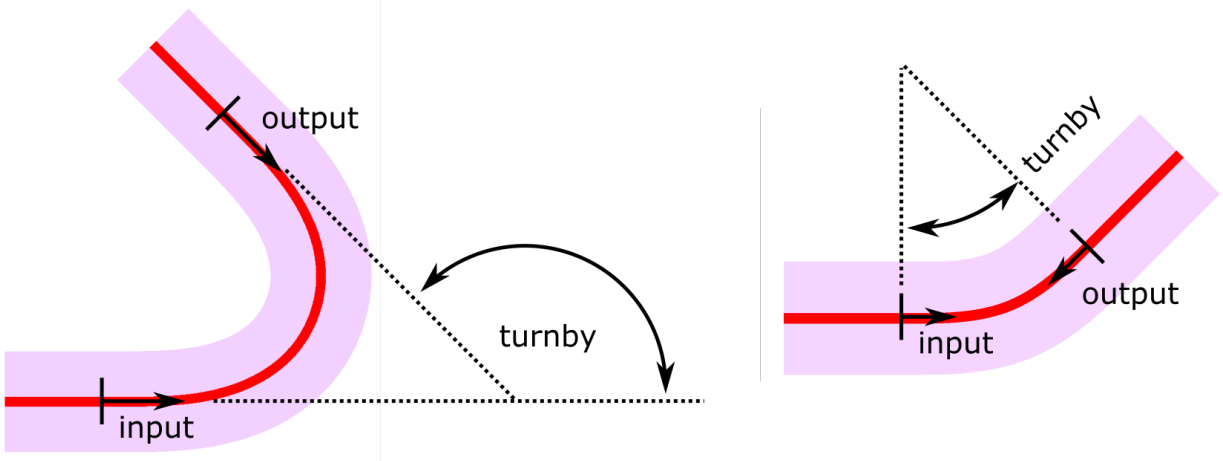
- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the taper, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.

direction = None
End protected variables



Sinusoidal S-bends

class picwriter.components.**SBend**(*wgt*, *height*, *length*, *port=(0, 0)*, *direction=u'EAST'*)
Sinusoidal S-shaped Bend Cell class. Creates a sinusoidal waveguide bend that can be used in waveguide routing. The number of points is computed based on the waveguide template grid resolution to automatically minimize grid errors.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **height** (float): Height of the S-bend
- **length** (float): Length of the S-bend

Keyword Args:

- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

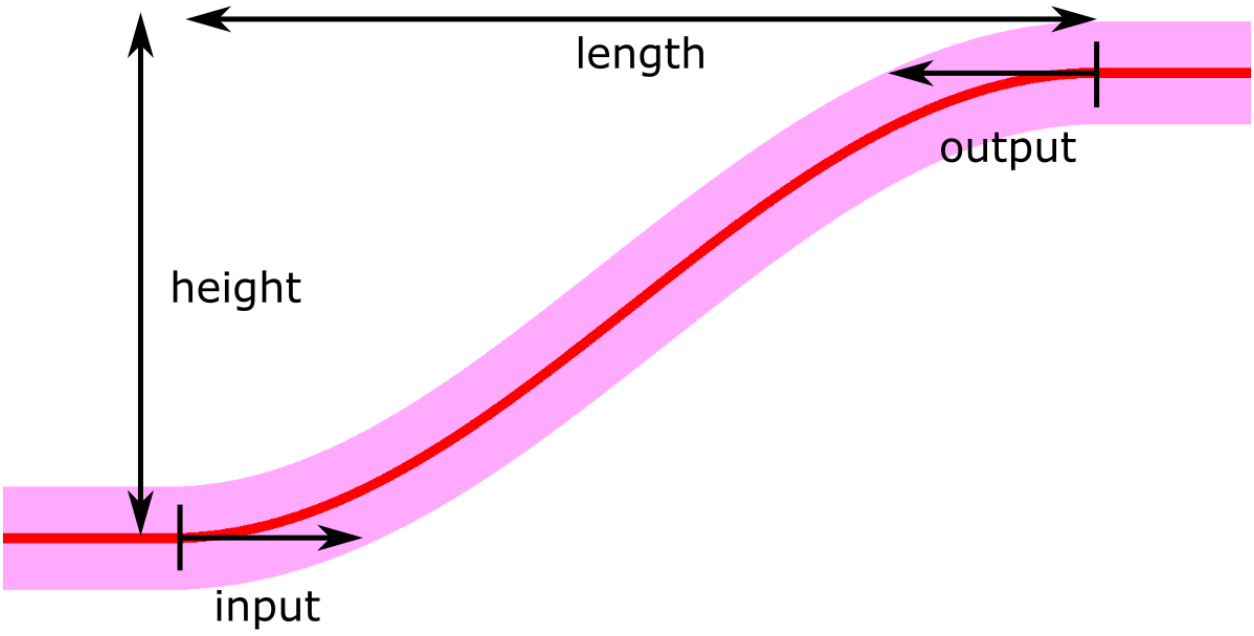
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the taper, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Mode converters

Linear tapers

Below is a standard taper class that can be used to linearly taper the waveguide width (such as for inverse tapers commonly used for fiber-to-chip coupling).

class picwriter.components.**Taper**(*wgt*, *length*, *end_width*, *end_clad_width=None*, *extra_clad_length=None*, *port=(0, 0)*, *direction=u'EAST'*)

Taper Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **length** (float): Length of the taper
- **end_width** (float): Final width of the taper (initial width received from WaveguieTemplate)

Keyword Args:

- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)
- **end_clad_width** (float): Clad width at the end of the taper. Defaults to the regular clad width.
- **extra_clad_length** (float): Extra cladding beyond the end of the taper. Defaults to 2*end_clad_width.

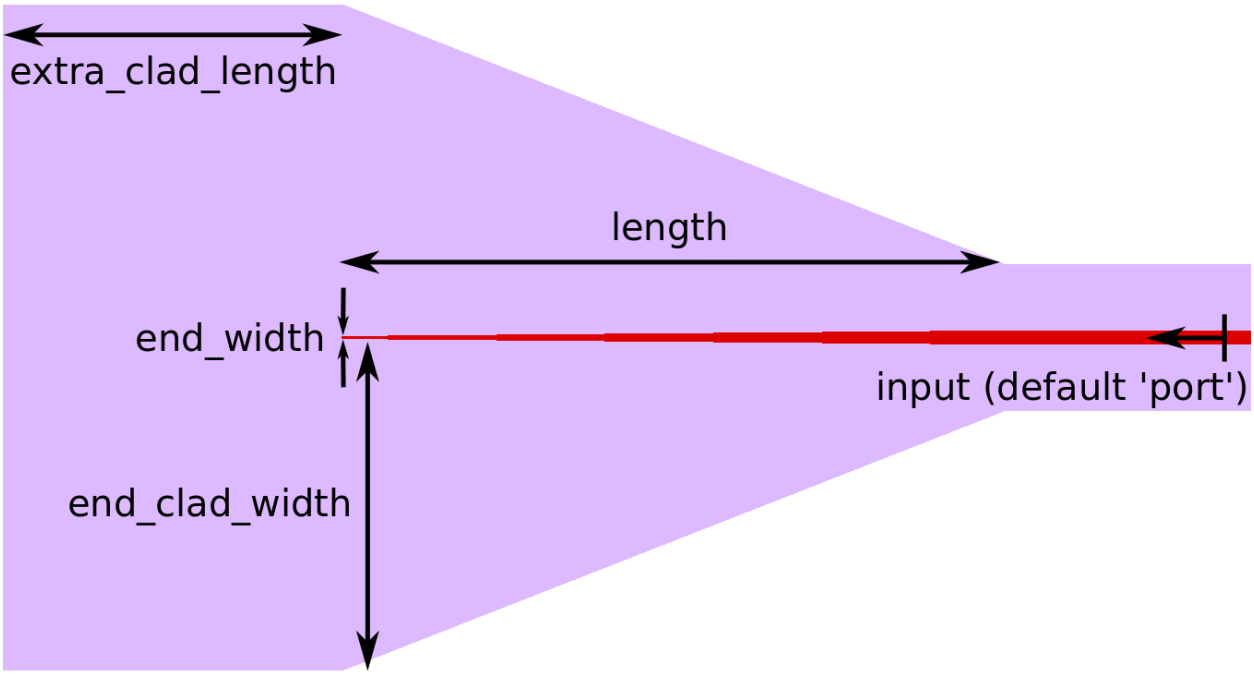
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the taper, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Strip-to-slot mode converters

class picwriter.components.**StripSlotConverter**(*wgt_input*, *wgt_output*, *length1*, *length2*, *start_rail_width*, *end_strip_width*, *d*, *input_strip=None*, *port=(0, 0)*, *direction=u'EAST'*)

Strip-to-Slot Side Converter Cell class. Adiabatically transforms a strip to a slot waveguide mode, with two sections. Section 1 introduces a narrow waveguide alongside the input strip waveguide and gradually lowers the gap between the strip waveguide and narrow side waveguide. Section 2 gradually converts the widths of the two waveguides until they are equal to the slot rail widths.

Args:

- **wgt_input** (WaveguideTemplate): WaveguideTemplate object for the input waveguide (should be either of type *strip* or *slot*).
- **wgt_output** (WaveguideTemplate): WaveguideTemplate object for the output waveguide (should be either of type *strip* or *slot*, opposite of the input type).
- **length1** (float): Length of section 1 that gradually changes the distance between the two waveguides.
- **length2** (float): Length of section 2 that gradually changes the widths of the two waveguides until equal to the slot waveguide rail widths.
- **start_rail_width** (float): Width of the narrow waveguide appearing next to the strip waveguide.
- **end_strip_width** (float): Width of the strip waveguide at the end of *length1* and before *length2*
- **d** (float): Distance between the outer edge of the strip waveguide and the start of the slot waveguide rail.

Keyword Args:

- **input_strip** (Boolean): If *True*, sets the input port to be the strip waveguide side. If *False*, slot waveguide is on the input. Defaults to *None*, in which case the input port waveguide template is used to choose.
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

Members:

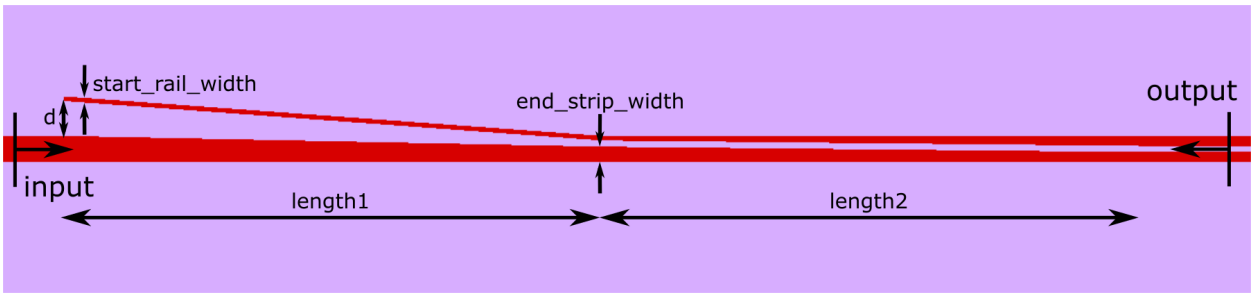
- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the taper, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.

Note: The waveguide and cladding layer/datatype are taken from the *wgt_slot* by default.



class picwriter.components.**StripSlotMMIConverter**(*wgt_input*, *wgt_output*, *w_mmi*, *l_mmi*, *length*, *input_strip=None*, *port=(0, 0)*, *direction=u'EAST'*)

Strip-to-Slot MMI Converter Cell class. For more information on this specific type of strip to slot mode converter, please see the original papers at https://doi.org/10.1364/OL.39.005665 and https://doi.org/10.1364/OE.24.007347.

Args:

- **wgt_input** (WaveguideTemplate): WaveguideTemplate object for the input waveguide (should be either of type *strip* or *slot*).
- **wgt_output** (WaveguideTemplate): WaveguideTemplate object for the output waveguide (should be either of type *strip* or *slot*, opposite of the input type).
- **w_mmi** (float): Width of the MMI region.
- **l_mmi** (float): Length of the MMI region.
- **length** (float): Length of the entire mode converter (MMI region + tapered region on slot waveguide side).

Keyword Args:

- **input_strip** (Boolean): If *True*, sets the input port to be the strip waveguide side. If *False*, slot waveguide is on the input. Defaults to *None*, in which case the input port waveguide template is used to choose.
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

Members:

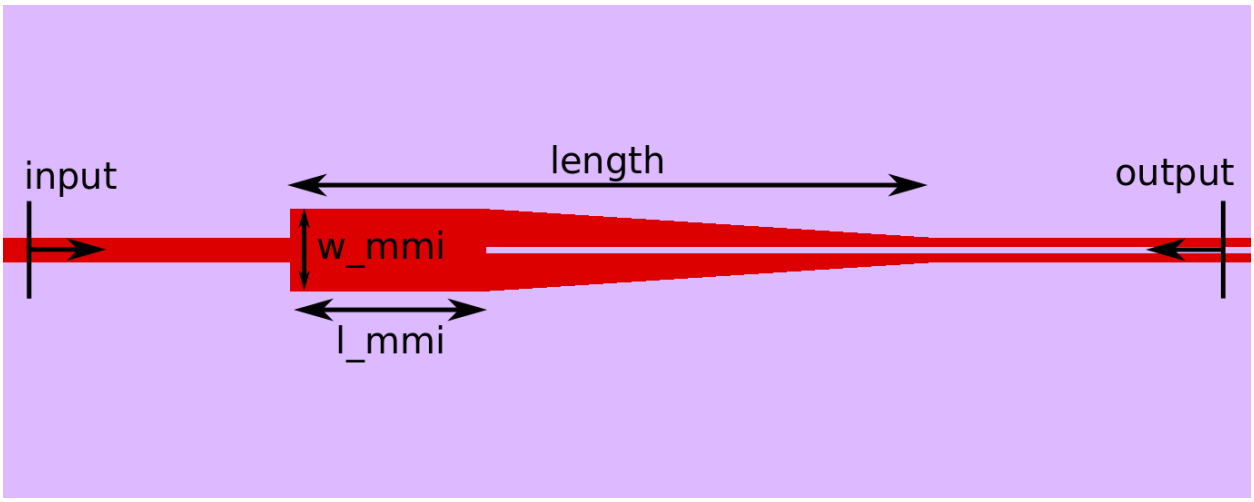
- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the taper, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.

Note: The waveguide and cladding layer/datatype are taken from the *wgt_slot* by default.



class picwriter.components.**StripSlotYConverter**(*wgt_input*, *wgt_output*, *length*, *d*, *end_strip_width=0*, *end_slot_width=0*, *input_strip=None*, *port=(0, 0)*, *direction=u'EAST'*)

Strip-to-Slot Y Converter Cell class. For more information on this specific type of strip to slot mode converter, please see the original paper at https://doi.org/10.1364/OL.34.001498.

Args:

- **wgt_input** (WaveguideTemplate): WaveguideTemplate object for the input waveguide (should be either of type *strip* or *slot*).
- **wgt_output** (WaveguideTemplate): WaveguideTemplate object for the output waveguide (should be either of type *strip* or *slot*, opposite of the input type).
- **length** (float): Length of the tapered region.
- **d** (float): Distance between the outer edge of the strip waveguide and the start of the slot waveguide rail.

Keyword Args:

- **end_strip_width** (float): End width of the strip waveguide (at the narrow tip). Defaults to 0.
- **end_slot_width** (float): End width of the slot waveguide (at the narrow tip). Defaults to 0.
- **input_strip** (Boolean): If *True*, sets the input port to be the strip waveguide side. If *False*, slot waveguide is on the input. Defaults to *None*, in which case the input port waveguide template is used to choose.
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

Members:

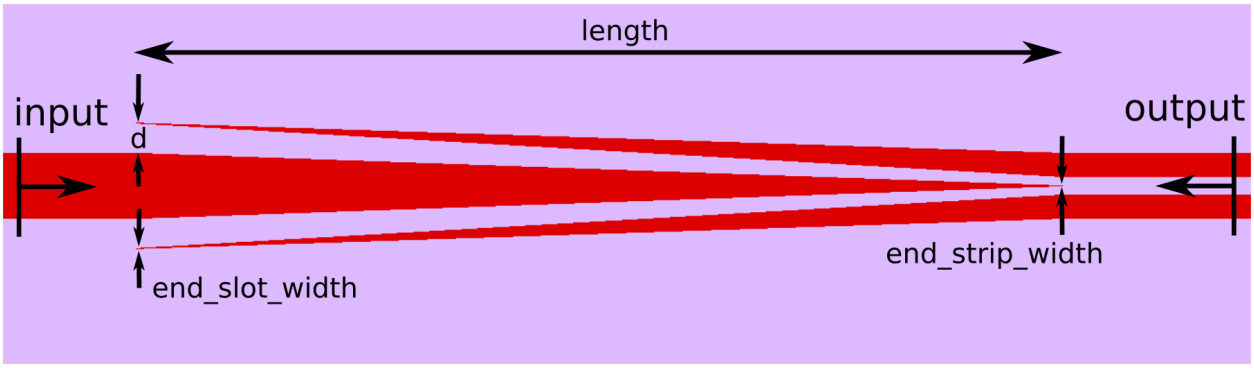
- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the taper, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.

Note: The waveguide and cladding layer/datatype are taken from the *wgt_slot* by default.



Grating Couplers

Grating Couplers

class picwriter.components.**GratingCoupler**(*wgt*, *theta=0.7853981633974483*, *length=30.0*, *taper_length=10.0*, *period=1.0*, *dutycycle=0.7*, *ridge=False*, *ridge_layers=(3, 0)*, *teeth_list=None*, *port=(0, 0)*, *direction=u'EAST'*)

Typical Grating Coupler Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object

Keyword Args:

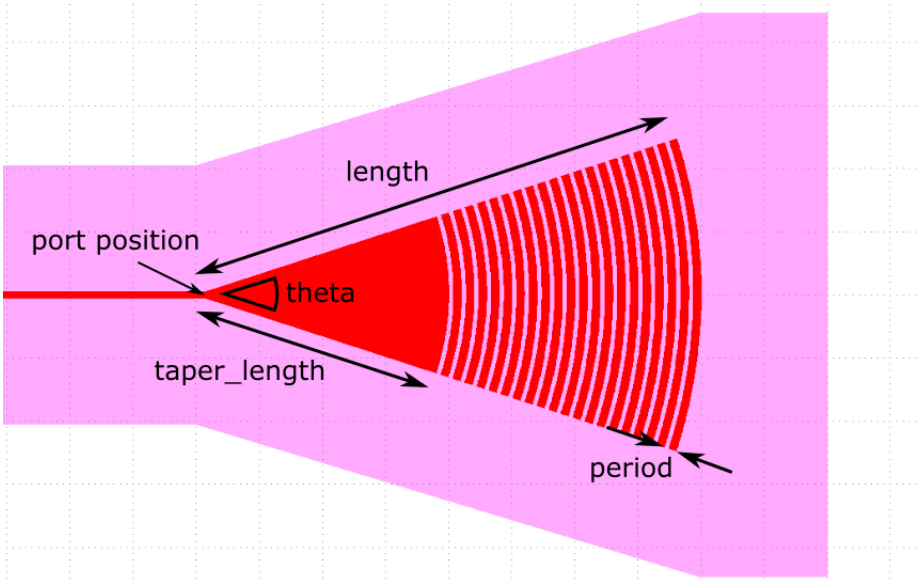
- **port** (tuple): Cartesian coordinate of the input port
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)
- **theta** (float): Angle of the waveguide. Defaults to pi/4.0
- **length** (float): Length of the total grating coupler region, measured from the output port. Defaults to 30.0
- **taper_length** (float): Length of the taper before the grating coupler. Defaults to 10.0
- **period** (float): Grating period. Defaults to 1.0
- **dutycycle** (float): dutycycle, determines the size of the 'gap' by dutycycle=(period-gap)/period. Defaults to 0.7
- **ridge** (boolean): If True, adds another layer to the grating coupler that can be used for partial etched gratings
- **ridge_layers** (tuple): Tuple specifying the layer/datatype of the ridge region. Defaults to (3,0)
- **teeth_list** (list): Can optionally pass a list of (gap, width) tuples to be used as the gap and teeth widths for irregularly spaced gratings. For example, [(0.6, 0.2), (0.7, 0.3), . . .] would be a gap of 0.6, then

a tooth of width 0.2, then gap of 0.7 and tooth of 0.3, and so on. Overrides *period*, *dutycycle*, and *length*. Defaults to None.

Members: **portlist** (dict): Dictionary with the relevant port information

Portlist format: portlist['output'] = {'port': (x1,y1), 'direction': 'dir1'}

Where in the above (x1,y1) is the same as the 'port' input, and 'dir1' is of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Straight Grating Couplers

class picwriter.components.**GratingCouplerStraight**(*wgt*, *port=(0, 0)*, *direction=u'EAST'*, *width=20*, *length=50*, *taper_length=20*, *period=1.0*, *dutycycle=0.5*)

Straight Grating Coupler Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object

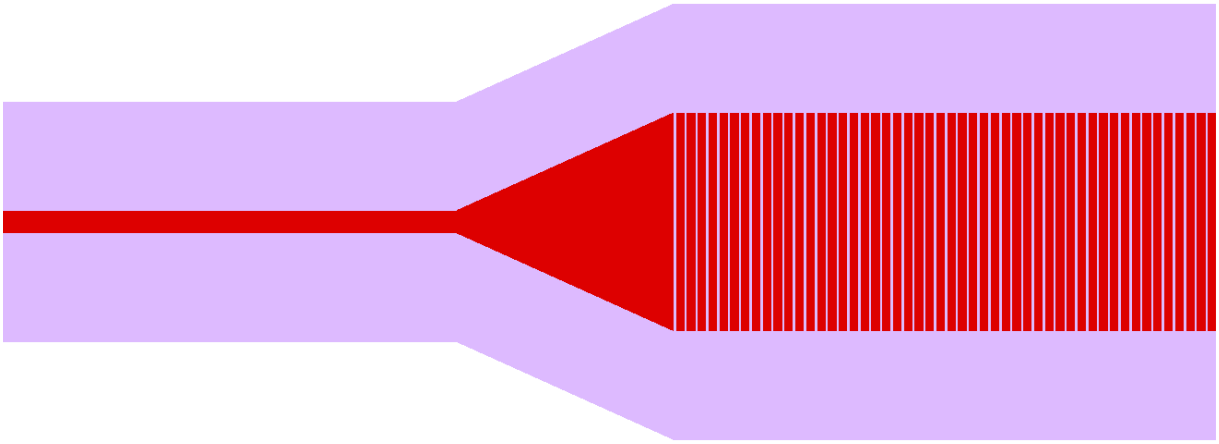
Keyword Args:

- **port** (tuple): Cartesian coordinate of the input port
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)
- **width** (float): Width of the grating region
- **length** (float): Length of the grating region
- **taper_length** (float): Length of the taper before the grating coupler
- **period** (float): Grating period
- **dutycycle** (float): dutycycle, determines the size of the 'gap' by dutycycle=(period-gap)/period.

Members: **portlist** (dict): Dictionary with the relevant port information

Portlist format: portlist['output'] = {'port': (x1,y1), 'direction': 'dir1'}

Where in the above (x1,y1) is the same as the 'port' input, and 'dir1' is of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Focusing Grating Couplers

class picwriter.components.**GratingCouplerFocusing**(*wgt*, *port=(0, 0)*, *direction=u'EAST'*, *focus_distance=None*, *width=20*, *length=50*, *period=1.0*, *dutycycle=0.5*, *wavelength=1.55*, *sin_theta=0.13917310096006544*, *evaluations=99*)

Standard Focusing Grating Coupler Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object

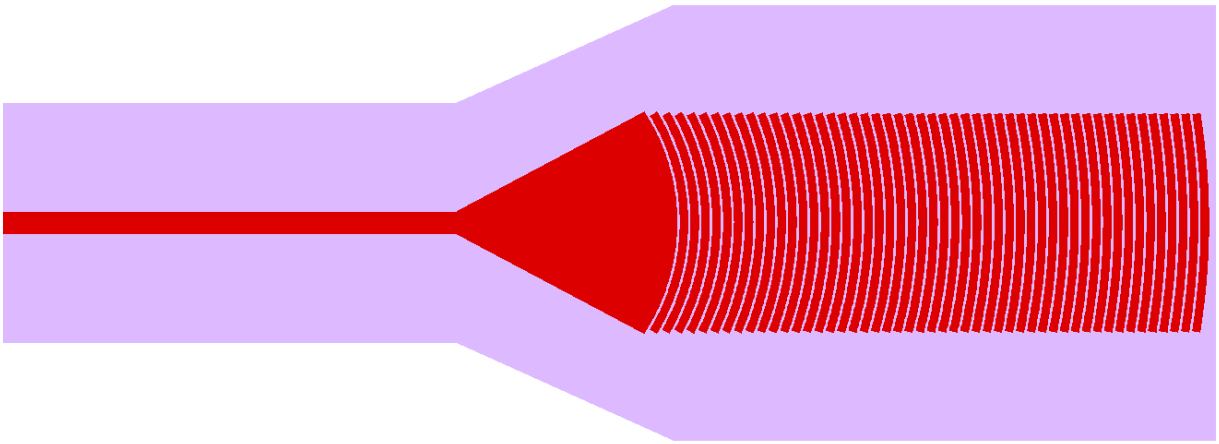
Keyword Args:

- **port** (tuple): Cartesian coordinate of the input port
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)
- **focus_distance** (float): Distance over which the light is focused to the waveguide port
- **width** (float): Width of the grating region
- **length** (float): Length of the grating region
- **period** (float): Grating period
- **dutycycle** (float): dutycycle, determines the size of the 'gap' by dutycycle=(period-gap)/period.
- **wavelength** (float): free space wavelength of the light
- **sin_theta** (float): sine of the incident angle
- **evaluations** (int): number of parameteric evaluations of path.parametric

Members: **portlist** (dict): Dictionary with the relevant port information

Portlist format: portlist['output'] = {'port': (x1,y1), 'direction': 'dir1'}

Where in the above (x1,y1) is the same as the 'port' input, and 'dir1' is of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Spirals

class picwriter.components.**Spiral**(*wgt*, *width*, *length*, *spacing=None*, *parity=1*, *port=(0, 0)*, *direction=u'NORTH'*)

Spiral Waveguide Cell class. The desired length of the spiral is first set, along with the spacing between input and output (the 'width' paramter). Then, the corresponding height of the spiral is automatically set.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **width** (float): width of the spiral (i.e. distance between input/output ports)
- **length** (float): desired length of the waveguide

Keyword Args:

- **spacing** (float): distance between parallel waveguides
- **parity** (int): If 1 spiral on right side, if -1 spiral on left side (mirror flip)
- **port** (tuple): Cartesian coordinate of the input port
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

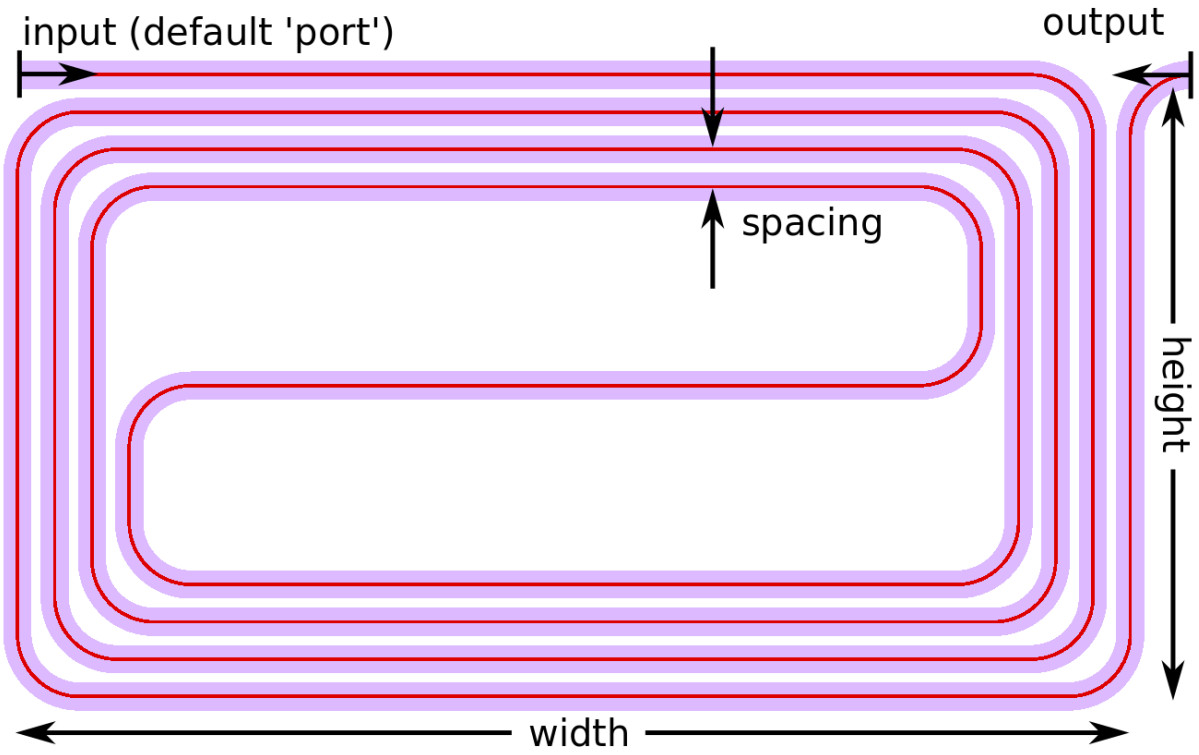
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) are the first elements of the spiral trace, (x2, y2) are the last elements of the spiral trace, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Waveguide Couplers

Directional Coupler

class picwriter.components.**DirectionalCoupler**(*wgt*, *length*, *gap*, *angle=0.5235987755982988*, *parity=1*, *port=(0, 0)*, *direction=u'EAST'*)

Directional Coupler Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **length** (float): Length of the coupling region.
- **gap** (float): Distance between the two waveguides.

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6.
- **parity** (integer -1 or 1): If -1, mirror-flips the structure so that the input port is actually the *bottom* port. Default = 1.
- **port** (tuple): Cartesian coordinate of the input port (AT TOP if parity=1, AT BOTTOM if parity=-1). Defaults to (0,0).

- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to 'EAST'.

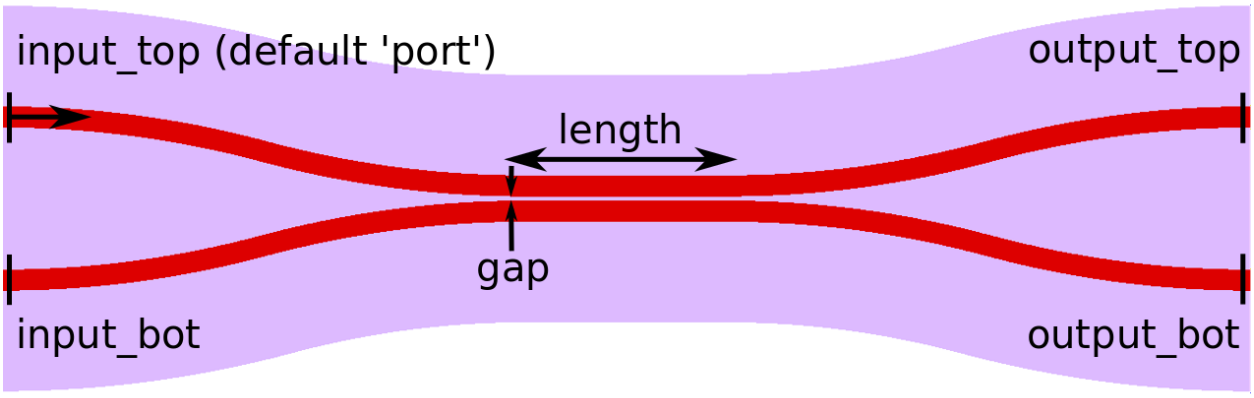
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input_top'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['input_bot'] = {'port': (x2,y2), 'direction': 'dir1'}
- portlist['output_top'] = {'port': (x3, y3), 'direction': 'dir3'}
- portlist['output_bot'] = {'port': (x4, y4), 'direction': 'dir4'}

Where in the above (x1,y1) (or (x2,y2) if parity=-1) is the same as the input 'port', (x3, y3), and (x4, y4) are the two output port locations. Directions 'dir1', 'dir2', etc. are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Adiabatic 3dB Coupler

class picwriter.components.**AdiabaticCoupler**(*wgt*, *length1*, *length2*, *gap*, *fargap*, *dw*, *angle=0.5235987755982988*, *port=(0, 0)*, *direction=u'EAST'*)

Adiabatic Coupler Cell class. Design based on asymmetric adiabatic 3dB coupler designs, such as those from https://doi.org/10.1364/CLEO.2010.CThAA2, https://doi.org/10.1364/CLEO_SI.2017.SF1I.5, and https://doi.org/10.1364/CLEO_SI.2018.STh4B.4.

In this design, Region I is the first half of the input S-bend waveguide where the input waveguides widths taper by +dw and -dw, Region II is the second half of the S-bend waveguide with constant, unbalanced widths, Region III is the region where the two asymmetric waveguides gradually come together, Region IV is the coupling region where the waveguides taper back to the original width at a fixed distance from one another, and Region IV is the output S-bend waveguide.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **length1** (float): Length of the region that gradually brings the two assymetric waveguides together
- **length2** (float): Length of the coupling region, where the asymmetric waveguides gradually become the same width.

- **gap** (float): Distance between the two waveguides.
- **fargap** (float): Largest distance between the two waveguides (Region III).
- **dw** (float): Change in waveguide width. Top arm tapers to the waveguide width - dw, bottom taper to width - dw.

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6.
- **port** (tuple): Cartesian coordinate of the input port (top left). Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to 'EAST'.

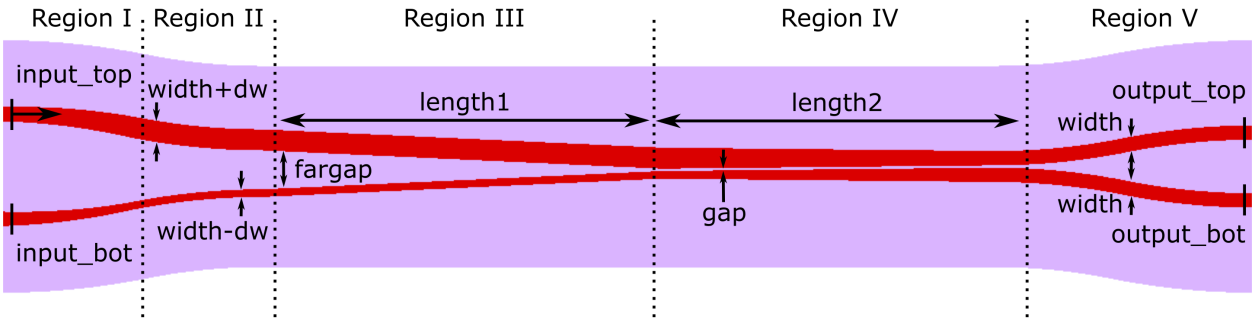
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input_top'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['input_bot'] = {'port': (x2,y2), 'direction': 'dir1'}
- portlist['output_top'] = {'port': (x3, y3), 'direction': 'dir3'}
- portlist['output_bot'] = {'port': (x4, y4), 'direction': 'dir4'}

Where in the above (x1,y1) is the same as the input 'port', (x3, y3), and (x4, y4) are the two output port locations. Directions 'dir1', 'dir2', etc. are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Adiabatic Full Coupler

class picwriter.components.**FullCoupler**(*wgt*, *length*, *gap*, *dw*, *angle=0.5235987755982988*, *parity=1*, *port=(0, 0)*, *direction=u'EAST'*)

Adiabatic Full Cell class. Design based on asymmetric adiabatic full coupler designs, such as the one reported in 'Integrated Optic Adiabatic Devices on Silicon' by Y. Shani, et al (IEEE Journal of Quantum Electronics, Vol. 27, No. 3 March 1991).

In this design, Region I is the first half of the input S-bend waveguide where the input waveguides widths taper by +dw and -dw, Region II is the second half of the S-bend waveguide with constant, unbalanced widths, Region III is the coupling region where the waveguides from unbalanced widths to balanced widths to reverse polarity unbalanced widths, Region IV is the fixed width waveguide that curves away from the coupling region, and

Region V is the final curve where the waveguides taper back to the regular width specified in the waveguide template.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **length** (float): Length of the coupling region.
- **gap** (float): Distance between the two waveguides.
- **dw** (float): Change in waveguide width. Top arm tapers to the waveguide width - dw, bottom taper to width - dw.

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6.
- **parity** (integer -1 or 1): If -1, mirror-flips the structure so that the input port is actually the *bottom* port. Default = 1.
- **port** (tuple): Cartesian coordinate of the input port (AT TOP if parity=1, AT BOTTOM if parity=-1). Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to 'EAST'.

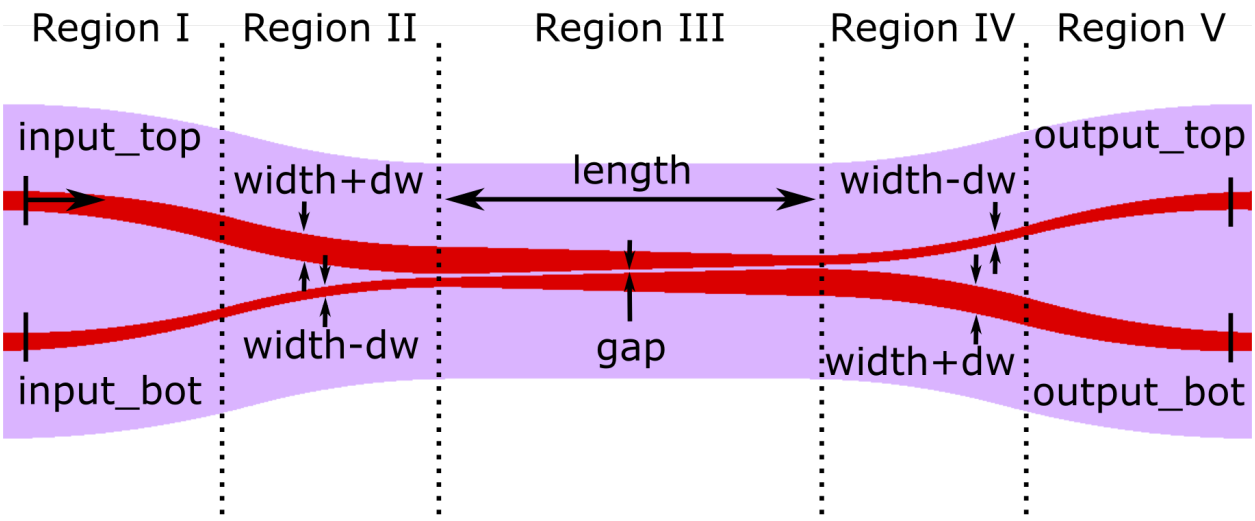
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input_top'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['input_bot'] = {'port': (x2,y2), 'direction': 'dir1'}
- portlist['output_top'] = {'port': (x3, y3), 'direction': 'dir3'}
- portlist['output_bot'] = {'port': (x4, y4), 'direction': 'dir4'}

Where in the above (x1,y1) (or (x2,y2) if parity=-1) is the same as the input 'port', (x3, y3), and (x4, y4) are the two output port locations. Directions 'dir1', 'dir2', etc. are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Sub-wavelength Grating Assisted Contra-Directional Coupler

For more details on the principles and operation behind this type of component, please see https://doi.org/10.1364/OE.25.025310. This component one regular waveguide (at top) and one sub-wavelength grating (at bottom) to selectively reflect a certain spectral band to the *intput_bot* port. Apodization of the top waveguide is also supported.

class picwriter.components.**SWGContraDirectionalCoupler**(*wgt*, *length*, *gap*, *period*, *dc*, *taper_length*, *w_phc_bot*, *top_angle=0.5235987755982988*, *width_top=None*, *width_bot=None*, *extra_swg_length=0.0*, *input_bot=False*, *apodization_top=False*, *apodization_far_dist=1.0*, *apodization_curv=None*, *fins=False*, *fin_size=(0.2, 0.05)*, *contradc_wgt=None*, *port=(0, 0)*, *direction=u'EAST'*)

SWG Contra-Directional Coupler Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **length** (float): Length of the coupling region.
- **gap** (float): Distance between the two waveguides.
- **period** (float): Period of the grating.
- **dc** (float): Duty cycle of the grating. Must be between 0 and 1.
- **taper_length** (float): Length of the taper region
- **w_phc_bot** (float): Width of the thin section of the bottom waveguide. w_phc_bot = 0 corresponds to disconnected periodic blocks.

Keyword Args:

- **top_angle** (float): Angle in radians (between 0 and pi/2) at which the *top* waveguide bends towards the coupling region. Default=pi/6.
- **width_top** (float): Width of the top waveguide in the coupling region. Defaults to the WaveguideTemplate wg width.
- **width_bot** (float): Width of the bottom waveguide in the coupling region. Defaults to the WaveguideTemplate wg width.
- **extra_swg_length** (float): Extra length of SWG waveguide between coupling region and taper. Default=0.0.
- **input_bot** (boolean): If *True*, will make the default input the bottom waveguide (rather than the top). Default='False'
- **apodization_top** (boolean): If *True*, will apodize the *coupling_gap* distance for the top waveguide using a Gaussian profile.
- **apodization_far_dist** (float): If *apodization_top'='True*, then this parameter sets how far away the coupling gap *starts*. The minimum coupling gap is defined by *gap*. Defaults to 1um.
- **apodization_curv** (float): If *apodization_top'='True*, then this parameter sets the curvature for the Gaussian apodization. Defaults to (10.0/length)**2.
- **fins** (boolean): If *True*, adds fins to the input/output waveguides. In this case a different template for the component must be specified. This feature is useful when performing electron-beam lithography and using different beam currents for fine features (helps to reduce stitching errors). Defaults to *False*
- **fin_size** ((x,y) Tuple): Specifies the x- and y-size of the *fins*. Defaults to 200 nm x 50 nm
- **contradc_wgt** (WaveguideTemplate): If *fins* above is True, a WaveguideTemplate (contradc_wgt) must be specified. This defines the layertype / datatype of the ContraDC (which will be separate from the input/output waveguides). Defaults to *None*
- **port** (tuple): Cartesian coordinate of the input port (AT TOP if input_bot=False, AT BOTTOM if input_bot=True). Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to 'EAST'.

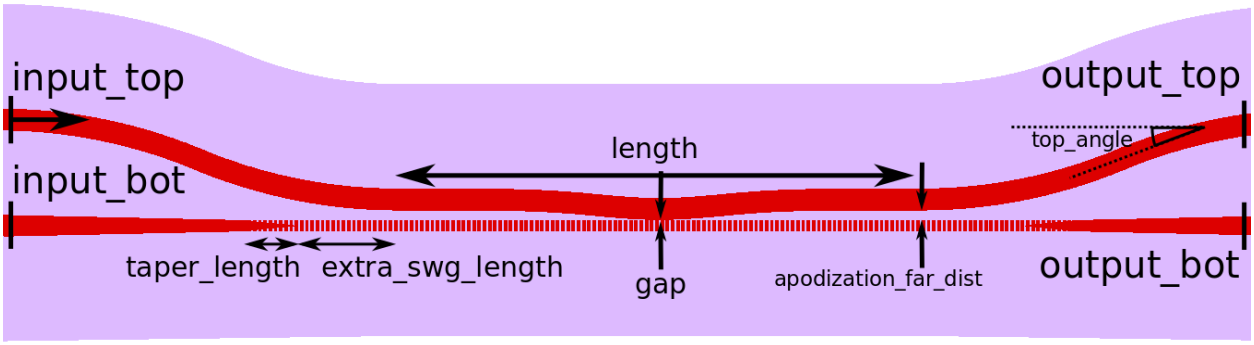
Members:

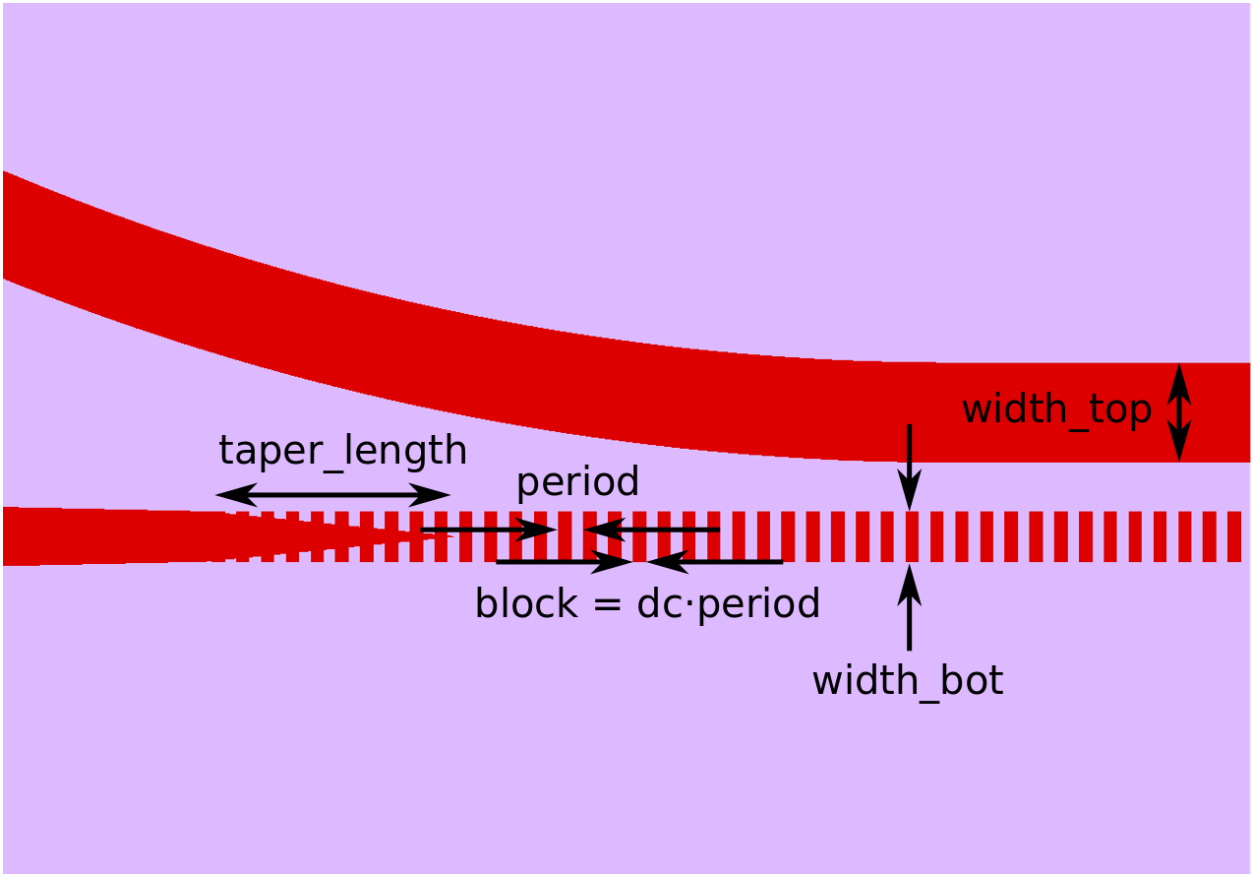
- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

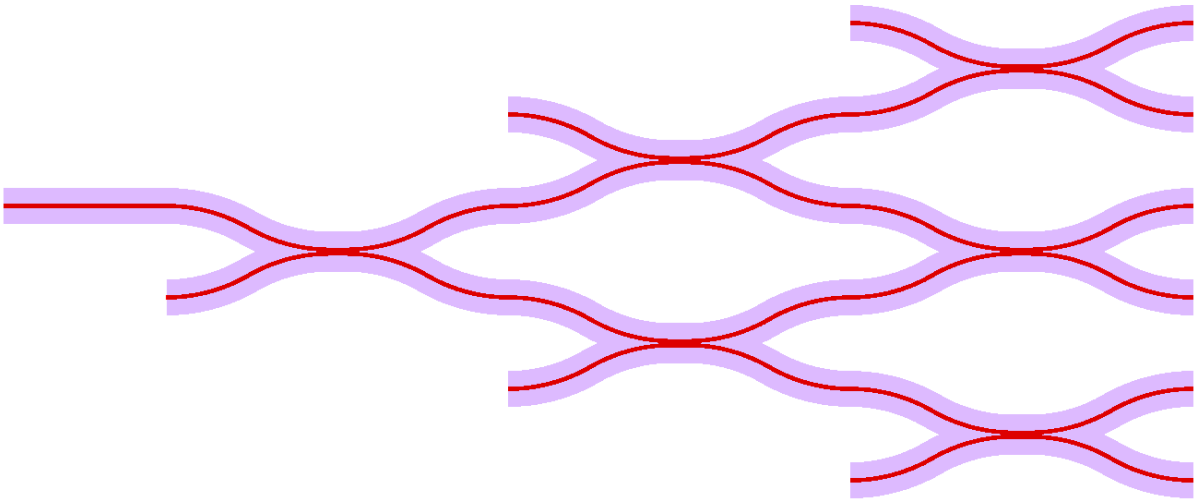
- portlist['input_top'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['input_bot'] = {'port': (x2,y2), 'direction': 'dir1'}
- portlist['output_top'] = {'port': (x3, y3), 'direction': 'dir3'}
- portlist['output_bot'] = {'port': (x4, y4), 'direction': 'dir4'}

Where in the above (x1,y1) (or (x2,y2) if input_bot=False) is the same as the input 'port', (x3, y3), and (x4, y4) are the two output port locations. Directions 'dir1', 'dir2', etc. are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.


input_top
input_bot
length
top_angle
output_top
output_bot
taper_length
extra_swg_length
gap
apodization_far_dist


width_top
taper_length
period
block = dc·period
width_bot

Example Usage



The directional coupler matrix shown above is generated by:

```
top = gdspy.Cell("top")
wgt = WaveguideTemplate(bend_radius=100, resist='+')

wg1=Waveguide([(0,0), (100,0)], wgt)
tk.add(top, wg1)

dc1 = DirectionalCoupler(wgt, 10.0, 0.5, angle=np.pi/6.0, parity=1, **wg1.portlist[
↪"output"])
dc2 = DirectionalCoupler(wgt, 10.0, 0.5, angle=np.pi/6.0, parity=-1, **dc1.portlist[
↪"output_top"])
dc3 = DirectionalCoupler(wgt, 10.0, 0.5, angle=np.pi/6.0, parity=1, **dc1.portlist[
↪"output_bot"])
dc4 = DirectionalCoupler(wgt, 10.0, 0.5, angle=np.pi/6.0, parity=1, **dc2.portlist[
↪"output_bot"])
dc5 = DirectionalCoupler(wgt, 10.0, 0.5, angle=np.pi/6.0, parity=-1, **dc2.portlist[
↪"output_top"])
dc6 = DirectionalCoupler(wgt, 10.0, 0.5, angle=np.pi/6.0, parity=1, **dc3.portlist[
↪"output_bot"])
tk.add(top, dc1)
tk.add(top, dc2)
tk.add(top, dc3)
tk.add(top, dc4)
tk.add(top, dc5)
tk.add(top, dc6)
```

Multi-Mode Interferometers (MMI's)

1x2 MMI

class picwriter.components.**MMI1x2**(*wgt*, *length*, *width*, *angle=0.5235987755982988*, *taper_width=None*, *taper_length=None*, *wg_sep=None*, *port=(0, 0)*, *direction=u'EAST'*)

1x2 multi-mode interfereomter (MMI) Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **length** (float): Length of the MMI region (along direction of propagation)
- **width** (float): Width of the MMI region (perpendicular to direction of propagation)

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6. Note: it is possible to generate a MMI with straight tapered outputs (not curved) by setting angle=0 and then connecting a straight Taper object to the desired MMI ports.
- **taper_width** (float): Maximum width of the taper region (default = wg_width from wg_template). Defaults to None (waveguide width).
- **taper_length** (float): Length of the input taper leading up to the MMI (single-port side). Defaults to None (taper_length=20).
- **wg_sep** (float): Separation between waveguides on the 2-port side (defaults to width/3.0). Defaults to None (width/3.0).
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

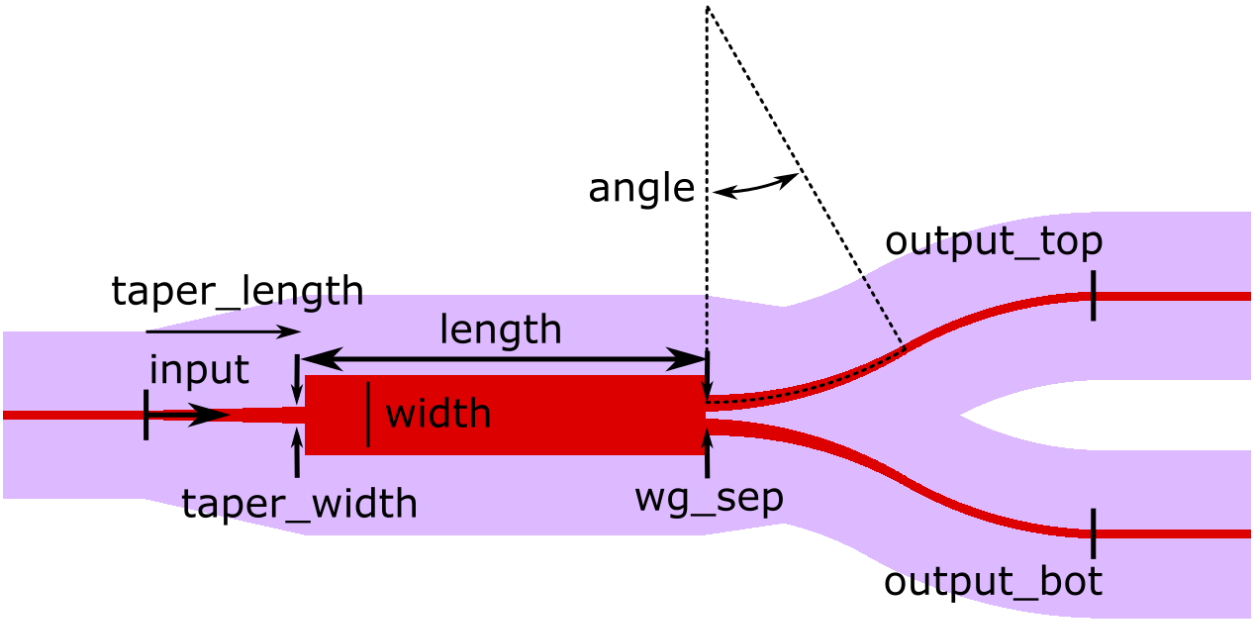
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output_top'] = {'port': (x2, y2), 'direction': 'dir2'}
- portlist['output_bot'] = {'port': (x3, y3), 'direction': 'dir3'}

Where in the above (x1,y1) is the input port, (x2, y2) is the top output port, (x3, y3) is the bottom output port, and 'dir1', 'dir2', 'dir3' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



2x2 MMI

class picwriter.components.**MMI2x2**(*wgt*, *length*, *width*, *angle=0.5235987755982988*, *taper_width=None*, *wg_sep=None*, *port=(0, 0)*, *direction=u'EAST'*)

2x2 multi-mode interferometer (MMI) Cell class. Two input ports, two output ports.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **length** (float): Length of the MMI region (along direction of propagation)
- **width** (float): Width of the MMI region (perpendicular to direction of propagation)

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6. Note: it is possible to generate a MMI with straight tapered outputs (not curved) by setting angle=0 and then connecting a straight Taper object to the desired MMI ports.
- **taper_width** (float): Maximum width of the taper region (default = wg_width from wg_template)
- **wg_sep** (float): Separation between waveguides on the 2-port side (defaults to width/3.0)
- **port** (tuple): Cartesian coordinate of the **top** input port
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

Members:

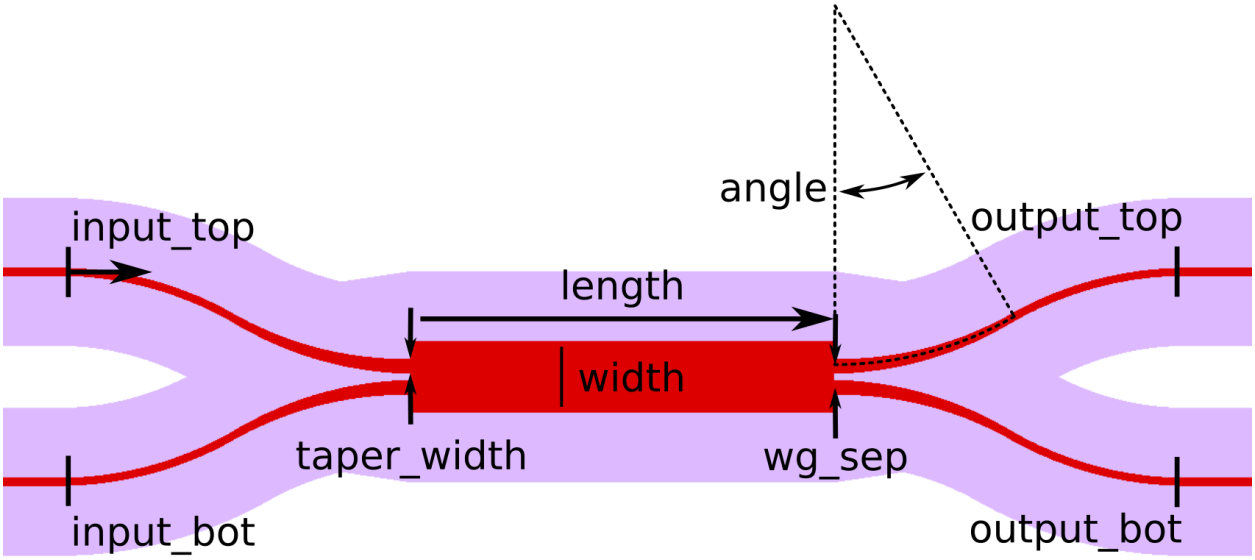
- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input_top'] = {'port': (x1,y1), 'direction': 'dir1'}

- portlist['input_bot'] = {'port': (x2, y2), 'direction': 'dir2'}
- portlist['output_top'] = {'port': (x3, y3), 'direction': 'dir3'}
- portlist['output_bot'] = {'port': (x4, y4), 'direction': 'dir4'}

Where in the above (x1,y1) is the input port, (x2, y2) is the top output port, (x3, y3) is the bottom output port, and 'dir1', 'dir2', 'dir3' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Cavities and Resonators

Ring Resonators

class picwriter.components.**Ring**(*wgt*, *radius*, *coupling_gap*, *wrap_angle=0*, *parity=1*, *draw_bus_wg=True*, *port=(0, 0)*, *direction=u'EAST'*)

Ring Resonator Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **radius** (float): Radius of the resonator
- **coupling_gap** (float): Distance between the bus waveguide and resonator

Keyword Args:

- **wrap_angle** (float): Angle in *radians* between 0 and pi that determines how much the bus waveguide wraps along the resonator. 0 corresponds to a straight bus waveguide, and pi corresponds to a bus waveguide wrapped around half of the resonator. Defaults to *0*.
- **parity** (1 or -1): If 1, resonator to left of bus waveguide, if -1 resonator to the right
- **port** (tuple): Cartesian coordinate of the input port (x1, y1)
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

- **draw_bus_wg** (bool): If *False*, does not generate the bus waveguide. Instead, the input/output port positions will be at the some location at the bottom of the ring, and the user can route their own bus waveguide. Defaults to *True*.

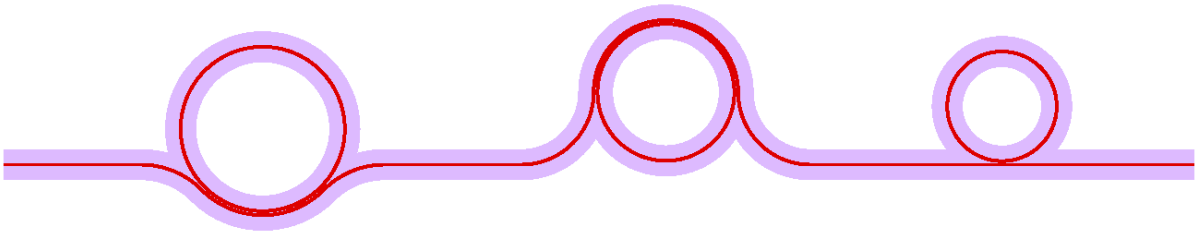
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the component, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



The code for generating the above three rings is:

```
top = gdspy.Cell("top")
wgt = WaveguideTemplate(bend_radius=50, resist='+')

wg1=Waveguide([(0,0), (100,0)], wgt)
tk.add(top, wg1)

r1 = Ring(wgt, 60.0, 1.0, wrap_angle=np.pi/2., parity=1, **wg1.portlist["output"])

wg2=Waveguide([r1.portlist["output"]["port"], (r1.portlist["output"]["port"][0]+100,␣
↪r1.portlist["output"]["port"][1])], wgt)
```

(continues on next page)

(continued from previous page)

```
tk.add(top, wg2)

r2 = Ring(wgt, 50.0, 0.8, wrap_angle=np.pi, parity=-1, **wg2.portlist["output"])

wg3=Waveguide([r2.portlist["output"]["port"], (r2.portlist["output"]["port"][0]+100,␣
↪r2.portlist["output"]["port"][1])], wgt)
tk.add(top, wg3)

r3 = Ring(wgt, 40.0, 0.6, parity=1, **wg3.portlist["output"])

wg4=Waveguide([r3.portlist["output"]["port"], (r3.portlist["output"]["port"][0]+100,␣
↪r3.portlist["output"]["port"][1])], wgt)
tk.add(top, wg4)

tk.add(top, r1)
tk.add(top, r2)
tk.add(top, r3)

gdspy.LayoutViewer()
```

Disk Resonators

class picwriter.components.**Disk**(*wgt*, *radius*, *coupling_gap*, *wrap_angle=0*, *parity=1*, *port=(0, 0)*, *direction=u'EAST'*)

Disk Resonator Cell class.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **radius** (float): Radius of the disk resonator
- **coupling_gap** (float): Distance between the bus waveguide and resonator

Keyword Args:

- **wrap_angle** (float): Angle in *radians* between 0 and pi (defaults to 0) that determines how much the bus waveguide wraps along the resonator. 0 corresponds to a straight bus waveguide, and pi corresponds to a bus waveguide wrapped around half of the resonator.
- **parity** (1 or -1): If 1, resonator to left of bus waveguide, if -1 resonator to the right
- **port** (tuple): Cartesian coordinate of the input port (x1, y1)
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

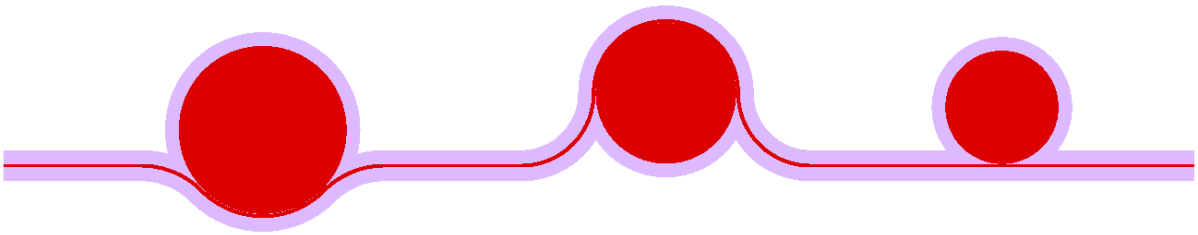
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the component, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



The code for generating the above three disks is:

```
top = gdspy.Cell("top")
wgt = WaveguideTemplate(bend_radius=50, resist='+')

wg1=Waveguide([(0,0), (100,0)], wgt)
tk.add(top, wg1)

r1 = Disk(wgt, 60.0, 1.0, wrap_angle=np.pi/2., parity=1, **wg1.portlist["output"])

wg2=Waveguide([r1.portlist["output"]["port"], (r1.portlist["output"]["port"][0]+100,␣
↪r1.portlist["output"]["port"][1])], wgt)
tk.add(top, wg2)

r2 = Disk(wgt, 50.0, 0.8, wrap_angle=np.pi, parity=-1, **wg2.portlist["output"])

wg3=Waveguide([r2.portlist["output"]["port"], (r2.portlist["output"]["port"][0]+100,␣
↪r2.portlist["output"]["port"][1])], wgt)
tk.add(top, wg3)

r3 = Disk(wgt, 40.0, 0.6, parity=1, **wg3.portlist["output"])

wg4=Waveguide([r3.portlist["output"]["port"], (r3.portlist["output"]["port"][0]+100,␣
↪r3.portlist["output"]["port"][1])], wgt)
tk.add(top, wg4)

tk.add(top, r1)
```

(continues on next page)

(continued from previous page)

```
tk.add(top, r2)
tk.add(top, r3)

gdspy.LayoutViewer()
```

Distributed Bragg Reflectors

class picwriter.components.**DBR**(*wgt*, *length*, *period*, *dc*, *w_phc*, *taper_length=20.0*, *fins=False*, *fin_size=(0.2, 0.05)*, *dbr_wgt=None*, *port=(0, 0)*, *direction=u'EAST'*)

Distributed Bragg Reflector Cell class. Tapers the input waveguide to a periodic waveguide structure with varying width (1-D photonic crystal).

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **length** (float): Length of the DBR region.
- **period** (float): Period of the repeated unit.
- **dc** (float): Duty cycle of the repeated unit (must be a float between 0 and 1.0).
- **w_phc** (float): Width of the thin section of the waveguide. w_phc = 0 corresponds to disconnected periodic blocks.

Keyword Args:

- **taper_length** (float): Length of the taper between the input/output waveguide and the DBR region. Defaults to 20.0.
- **fins** (boolean): If *True*, adds fins to the input/output waveguides. In this case a different template for the component must be specified. This feature is useful when performing electron-beam lithography and using different beam currents for fine features (helps to reduce stitching errors). Defaults to *False*
- **fin_size** ((x,y) Tuple): Specifies the x- and y-size of the *fins*. Defaults to 200 nm x 50 nm
- **dbr_wgt** (WaveguideTemplate): If *fins* above is True, a WaveguideTemplate (dbr_wgt) must be specified. This defines the layertype / datatype of the DBR (which will be separate from the input/output waveguides). Defaults to *None*
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians)

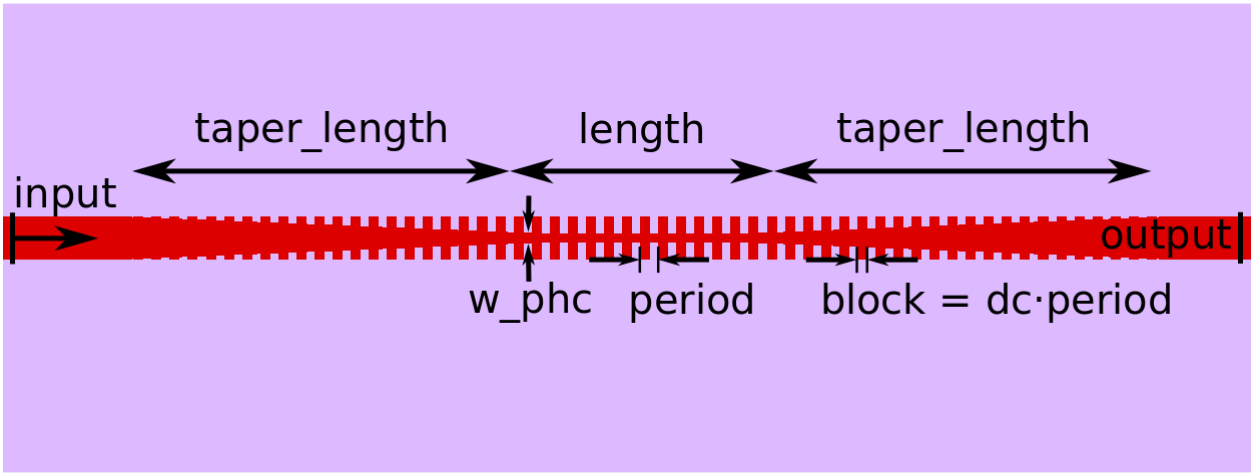
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) is the same as the 'port' input, (x2, y2) is the end of the DBR, and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it.



Mach-Zehnder Interferometers

Regular Mach-Zehnder

class picwriter.components.**MachZehnder**(*wgt*, *MMIlength*, *MMIwidth*, *angle=0.5235987755982988*, *MMItaper_width=None*, *MMItaper_length=None*, *MMIwg_sep=None*, *arm1=0*, *arm2=0*, *heater=False*, *heater_length=400*, *mt=None*, *port=(0, 0)*, *direction=u'EAST'*)

Mach-Zehnder Cell class with thermo-optic option. It is possible to generate your own Mach-Zehnder from the waveguide and MMI1x2 classes, but this class is simply a shorthand (with some extra type-checking). Defaults to a *balanced* Mach Zehnder.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **MMIlength** (float): Length of the 1x2 MMI region (along direction of propagation)
- **MMIwidth** (float): Width of the 1x2 MMI region (perpendicular to direction of propagation).

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6.
- **MMItaper_width** (float): Maximum width of the 1x2 MMI taper region (default = wg_width from wg_template). Defaults to None (waveguide width).
- **MMItaper_length** (float): Length of the taper leading up to the 1x2 MMI. Defaults to None (taper_length=20).

- **MMIwg_sep** (float): Separation between waveguides on the 2-port side of the 1x2 MMI (defaults to width/3.0)
- **arm1** (float): Additional length of the top arm (when going *'EAST'*). Defaults to zero.
- **arm2** (float): Additional length of the bottom arm (when going *'EAST'*). Defaults to zero.
- **heater** (boolean): If true, adds heater on-top of one MZI arm. Defaults to False.
- **heater_length** (float): Specifies the length of the heater along the waveguide. Doesn't include the length of the 180 degree bend. Defaults to 400.0.
- **mt** (MetalTemplate): If 'heater' is true, must specify a Metal Template that defines heater & heater cladding layers.
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to *'EAST'* (0 radians)

Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}
- portlist['heater_top_in'] = {'port', (x3, y3), 'direction': 'dir3'}
- portlist['heater_top_out'] = {'port', (x4, y4), 'direction': 'dir4'}
- portlist['heater_bot_in'] = {'port', (x5, y5), 'direction': 'dir5'}
- portlist['heater_bot_out'] = {'port', (x6, y6), 'direction': 'dir6'}

Where in the above (x1,y1) is the input port, (x2, y2) is the output port, and the directions are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it. Four additional ports are created for the heaters if the *heater* argument is True. Metals are not generated, but should be connected to the specified 'heater' ports.

Mach-Zehnder Switches

Same as the type above, but instead places a 1x2 MMI for the input, and a 2x2 MMI at the output. Tuning of the respective arms allows for optical switching between the two output paths.

class picwriter.components.**MachZehnderSwitch1x2**(*wgt*, *MMI1x2length*, *MMI1x2width*, *MMI2x2length*, *MMI2x2width*, *angle=0.5235987755982988*, *MMI1x2taper_width=None*, *MMI1x2taper_length=None*, *MMI1x2wg_sep=None*, *MMI2x2taper_width=None*, *MMI2x2wg_sep=None*, *arm1=0*, *arm2=0*, *heater=False*, *heater_length=400*, *mt=None*, *port=(0, 0)*, *direction=u'EAST'*)

Standard Mach-Zehnder Optical Switch Cell class with heaters on each arm. It is possible to generate your own Mach-Zehnder from the waveguide and MMI1x2 classes, but this class is simply a shorthand (with some extra type-checking). Defaults to a *balanced* Mach Zehnder.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **MMI1x2length** (float): Length of the 1x2 MMI region (along direction of propagation)
- **MMI1x2width** (float): Width of the 1x2 MMI region (perpendicular to direction of propagation).
- **MMI2x2length** (float): Length of the 2x2 MMI region (along direction of propagation)
- **MMI2x2width** (float): Width of the 2x2 MMI region (perpendicular to direction of propagation).

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6.
- **MMI1x2taper_width** (float): Maximum width of the 1x2 MMI taper region (default = wg_width from wg_template). Defaults to None (waveguide width).
- **MMI1x2taper_length** (float): Length of the taper leading up to the 1x2 MMI. Defaults to None (taper_length=20).
- **MMI1x2wg_sep** (float): Separation between waveguides on the 2-port side of the 1x2 MMI (defaults to width/3.0)
- **MMI2x2taper_width** (float): Maximum width of the 2x2 MMI taper region (default = wg_width from wg_template). Defaults to None (waveguide width).
- **MMI2x2wg_sep** (float): Separation between waveguides of the 2x2 MMI (defaults to width/3.0)
- **arm1** (float): Additional length of the top arm (when going *'EAST'*). Defaults to zero.
- **arm2** (float): Additional length of the bottom arm (when going *'EAST'*). Defaults to zero.
- **heater** (boolean): If true, adds heater on-top of one MZI arm. Defaults to False.
- **heater_length** (float): Specifies the length of the heater along the waveguide. Doesn't include the length of the 180 degree bend. Defaults to 400.0.
- **mt** (MetalTemplate): If 'heater' is true, must specify a Metal Template that defines heater & heater cladding layers.
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the taper will point *towards*, must be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to *'EAST'* (0 radians)

Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output_top'] = {'port': (x2, y2), 'direction': 'dir2'}
- portlist['output_bot'] = {'port': (x3, y3), 'direction': 'dir3'}
- portlist['heater_top_in'] = {'port', (x4, y4), 'direction': 'dir4'}
- portlist['heater_top_out'] = {'port', (x5, y5), 'direction': 'dir5'}
- portlist['heater_bot_in'] = {'port', (x6, y6), 'direction': 'dir6'}
- portlist['heater_bot_out'] = {'port', (x7, y7), 'direction': 'dir7'}

Where in the above (x1,y1) is the input port, (x2, y2) is the top output port, (x3, y3) is the bottom output port, and the directions are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it. Four additional ports are created for the heaters if the *heater* argument is True. Metals are not generated, but should be connected to the specified 'heater' ports.

class picwriter.components.**MachZehnderSwitchDC1x2**(*wgt*, *MMI1x2length*, *MMI1x2width*, *DClength*, *DCgap*, *angle=0.5235987755982988*, *MMI1x2taper_width=None*, *MMI1x2taper_length=None*, *MMI1x2wg_sep=None*, *arm1=0*, *arm2=0*, *heater=False*, *heater_length=400*, *mt=None*, *port=(0, 0)*, *direction=u'EAST'*)

Standard Mach-Zehnder Optical Switch Cell class with heaters on each arm and a directional coupler. It is possible to generate your own Mach-Zehnder from the other classes, but this class is simply a shorthand (with some extra type-checking). Defaults to a *balanced* Mach Zehnder.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **MMI1x2length** (float): Length of the 1x2 MMI region (along direction of propagation)
- **MMI1x2width** (float): Width of the 1x2 MMI region (perpendicular to direction of propagation).
- **DClength** (float): Length of the directional coupler region
- **DCgap** (float): Size of the directional coupler gap

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region (same for MMI & DC). Default=pi/6.
- **MMI1x2taper_width** (float): Maximum width of the 1x2 MMI taper region (default = wg_width from wg_template). Defaults to None (waveguide width).
- **MMI1x2taper_length** (float): Length of the taper leading up to the 1x2 MMI. Defaults to None (taper_length=20).
- **MMI1x2wg_sep** (float): Separation between waveguides on the 2-port side of the 1x2 MMI (defaults to width/3.0)
- **arm1** (float): Additional length of the top arm (when going *'EAST'*). Defaults to zero.
- **arm2** (float): Additional length of the bottom arm (when going *'EAST'*). Defaults to zero.
- **heater** (boolean): If true, adds heater on-top of one MZI arm. Defaults to False.
- **heater_length** (float): Specifies the length of the heater along the waveguide. Doesn't include the length of the 180 degree bend. Defaults to 400.0.
- **mt** (MetalTemplate): If 'heater' is true, must specify a Metal Template that defines heater & heater cladding layers.
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the taper will point *towards*, must be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to *'EAST'* (0 radians)

Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output_top'] = {'port': (x2, y2), 'direction': 'dir2'}
- portlist['output_bot'] = {'port': (x3, y3), 'direction': 'dir3'}
- portlist['heater_top_in'] = {'port', (x4, y4), 'direction': 'dir4'}
- portlist['heater_top_out'] = {'port', (x5, y5), 'direction': 'dir5'}
- portlist['heater_bot_in'] = {'port', (x6, y6), 'direction': 'dir6'}
- portlist['heater_bot_out'] = {'port', (x7, y7), 'direction': 'dir7'}

Where in the above (x1,y1) is the input port, (x2, y2) is the top output port, (x3, y3) is the bottom output port, and the directions are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it. Four additional ports are created for the heaters if the *heater* argument is True. Metals are not generated, but should be connected to the specified 'heater' ports.

class picwriter.components.**MachZehnderSwitchDC2x2**(*wgt*, *DC1length*, *DC1gap*, *DC2length*, *DC2gap*, *angle=0.5235987755982988*, *arm1=0*, *arm2=0*, *heater=False*, *heater_length=400*, *mt=None*, *port=(0, 0)*, *direction=u'EAST'*)

Standard Mach-Zehnder Optical Switch Cell class with heaters on each arm and a directional coupler for both input and output. It is possible to generate your own Mach-Zehnder from the other classes, but this class is simply a shorthand (with some extra type-checking). Defaults to a *balanced* Mach Zehnder.

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object
- **DC1length** (float): Length of the directional coupler region at the input
- **DC1gap** (float): Size of the directional coupler gap at the input
- **DC2length** (float): Length of the directional coupler region at the output
- **DC2gap** (float): Size of the directional coupler gap at the output

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6.
- **arm1** (float): Additional length of the top arm (when going *'EAST'*). Defaults to zero.
- **arm2** (float): Additional length of the bottom arm (when going *'EAST'*). Defaults to zero.
- **heater** (boolean): If true, adds heater on-top of one MZI arm. Defaults to False.
- **heater_length** (float): Specifies the length of the heater along the waveguide. Doesn't include the length of the 180 degree bend. Defaults to 400.0.
- **mt** (MetalTemplate): If 'heater' is true, must specify a Metal Template that defines heater & heater cladding layers.
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the taper will point *towards*, must be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to *'EAST'* (0 radians)

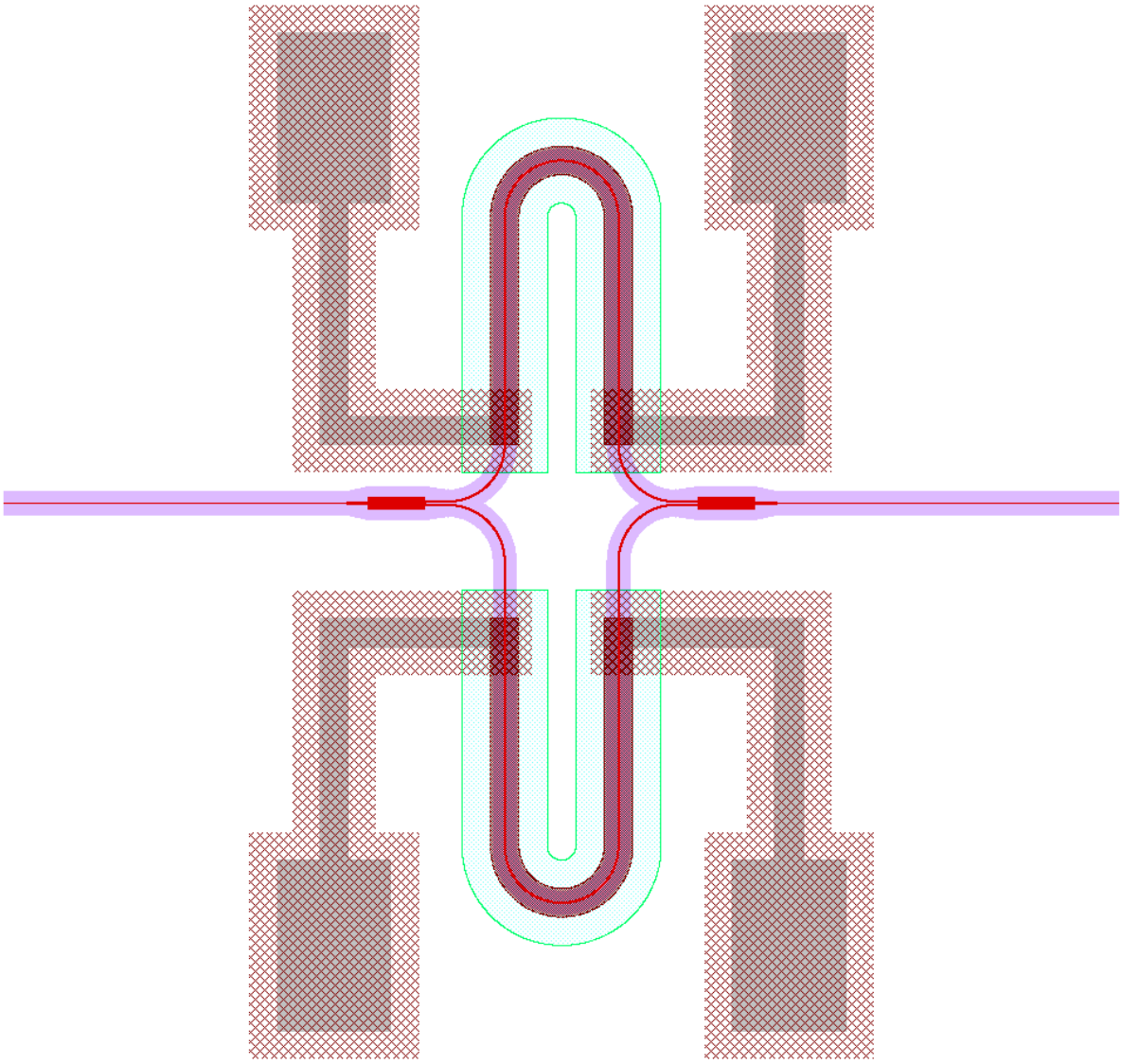
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input_top'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['input_bot'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output_top'] = {'port': (x2, y2), 'direction': 'dir2'}
- portlist['output_bot'] = {'port': (x3, y3), 'direction': 'dir3'}
- portlist['heater_top_in'] = {'port', (x4, y4), 'direction': 'dir4'}
- portlist['heater_top_out'] = {'port', (x5, y5), 'direction': 'dir5'}
- portlist['heater_bot_in'] = {'port', (x6, y6), 'direction': 'dir6'}
- portlist['heater_bot_out'] = {'port', (x7, y7), 'direction': 'dir7'}

Where in the above (x1,y1) is the input port, (x2, y2) is the top output port, (x3, y3) is the bottom output port, and the directions are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, *or* an angle in *radians*. 'Direction' points *towards* the waveguide that will connect to it. Four additional ports are created for the heaters if the *heater* argument is True. Metals are not generated, but should be connected to the specified 'heater' ports.



The above image of an unbalanced Mach-Zehnder with heaters for thermo-optic phase modulation is easily generated from the following code:

```
top = gdspy.Cell("top")
wgt = WaveguideTemplate(bend_radius=50, wg_width=1.0, resist='+')
htr_mt = MetalTemplate(width=25, clad_width=25, bend_radius=wgt.bend_radius, resist='+
↪', fab="ETCH", metal_layer=13, metal_datatype=0, clad_layer=14, clad_datatype=0)
mt = MetalTemplate(width=25, clad_width=25, resist='+', fab="ETCH", metal_layer=11,␣
↪metal_datatype=0, clad_layer=12, clad_datatype=0)

wg_in = Waveguide([(0,0), (300,0)], wgt)
tk.add(top, wg_in)
mzi = MachZehnder(wgt, MMIlength=50, MMIwidth=10, MMItaper_width=2.0, MMIwg_sep=3,␣
↪arm1=0, arm2=100, heater=True, heater_length=400, mt=htr_mt, **wg_in.portlist[
↪"output"])
tk.add(top, mzi)
wg_out = Waveguide([mzi.portlist["output"]["port"], (mzi.portlist["output"]["port
↪"][0]+300, mzi.portlist["output"]["port"][1])], wgt)
```

(continues on next page)

(continued from previous page)

```
tk.add(top, wg_out)

mt1=MetalRoute([mzi.portlist["heater_top_in"]["port"],
                (mzi.portlist["heater_top_in"]["port"][0]-150, mzi.portlist["heater_
↪top_in"]["port"][1]),
                (mzi.portlist["heater_top_in"]["port"][0]-150, mzi.portlist["heater_
↪top_in"]["port"][1]+200)], mt)
mt2=MetalRoute([mzi.portlist["heater_top_out"]["port"],
                (mzi.portlist["heater_top_out"]["port"][0]+150, mzi.portlist["heater_
↪top_out"]["port"][1]),
                (mzi.portlist["heater_top_out"]["port"][0]+150, mzi.portlist["heater_
↪top_out"]["port"][1]+200)], mt)
mt3=MetalRoute([mzi.portlist["heater_bot_in"]["port"],
                (mzi.portlist["heater_bot_in"]["port"][0]-150, mzi.portlist["heater_
↪bot_in"]["port"][1]),
                (mzi.portlist["heater_bot_in"]["port"][0]-150, mzi.portlist["heater_
↪bot_in"]["port"][1]-200)], mt)
mt4=MetalRoute([mzi.portlist["heater_bot_out"]["port"],
                (mzi.portlist["heater_bot_out"]["port"][0]+150, mzi.portlist["heater_
↪bot_out"]["port"][1]),
                (mzi.portlist["heater_bot_out"]["port"][0]+150, mzi.portlist["heater_
↪bot_out"]["port"][1]-200)], mt)
tk.add(top, mt1)
tk.add(top, mt2)
tk.add(top, mt3)
tk.add(top, mt4)
tk.add(top, Bondpad(mt, **mt1.portlist["output"]))
tk.add(top, Bondpad(mt, **mt2.portlist["output"]))
tk.add(top, Bondpad(mt, **mt3.portlist["output"]))
tk.add(top, Bondpad(mt, **mt4.portlist["output"]))
```

Alignment Markers

Alignment Cross

class picwriter.components.**AlignmentCross**(*cross_length*, *cross_width*, *small_cross_width=None*, *center=(0, 0)*, *layer=1*, *datatype=0*)

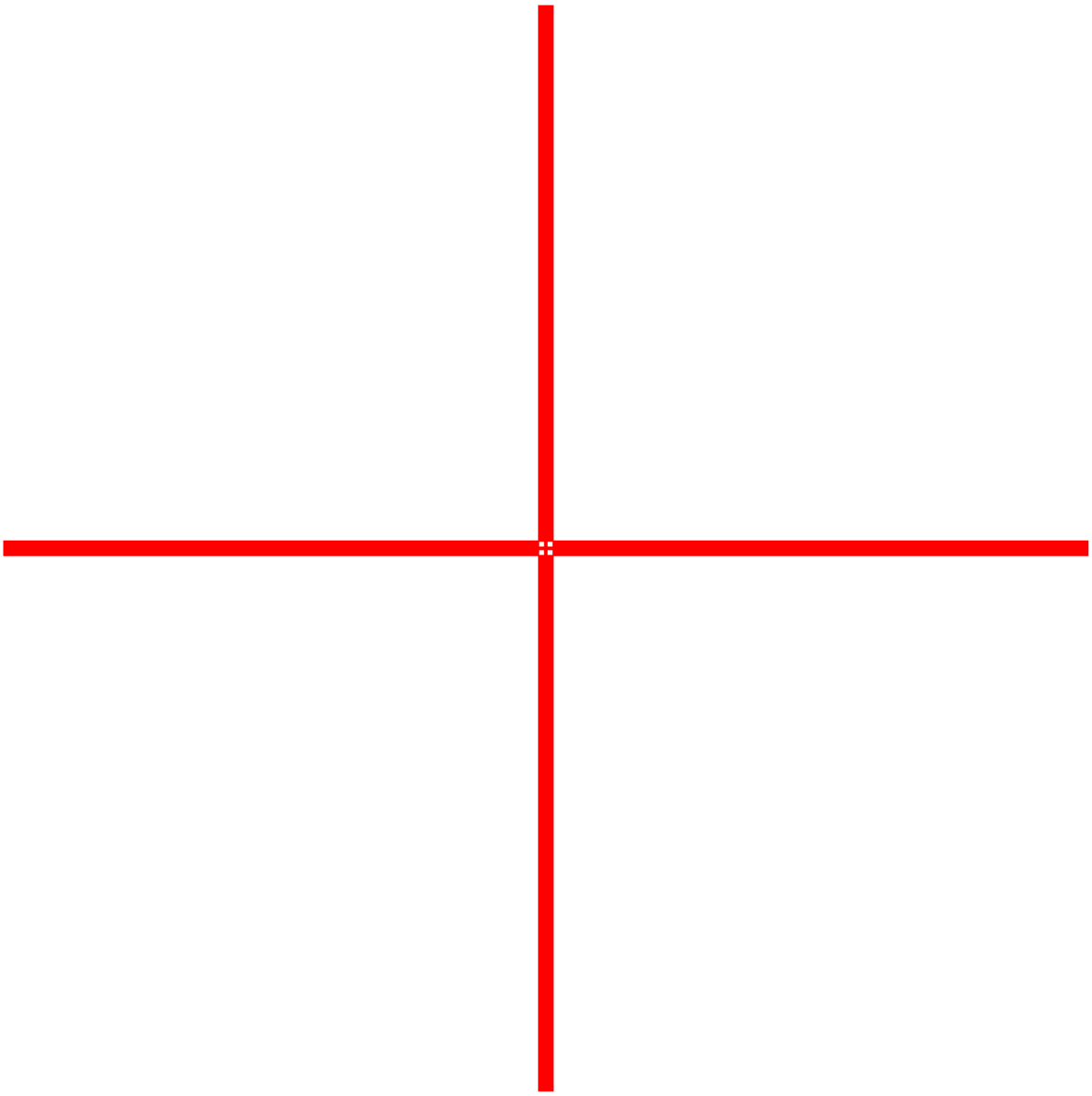
Cross Cell class, used for alignment

Args:

- **cross_length** (float): Length of each arm of the cross.
- **cross_width** (float): Width of the cross arm
- **center** (tuple): Coordinate (x1, y1) of the center of the cross

Keyword Args:

- **small_cross_width** (float): If given, sets the width of the small cross in the center of the big cross. Defaults to 1/4 the value of cross_width
- **layer** (int): Layer to place the marker on. Defaults to 1
- **datatype** (int): Datatype to place the marker on. Defaults to 0



Alignment Target

class picwriter.components.**AlignmentTarget**(*diameter*, *ring_width*, *num_rings=10*, *center=(0, 0)*, *layer=1*, *datatype=0*)

Standard Target Cell class, used for alignment. Set of concentric circles

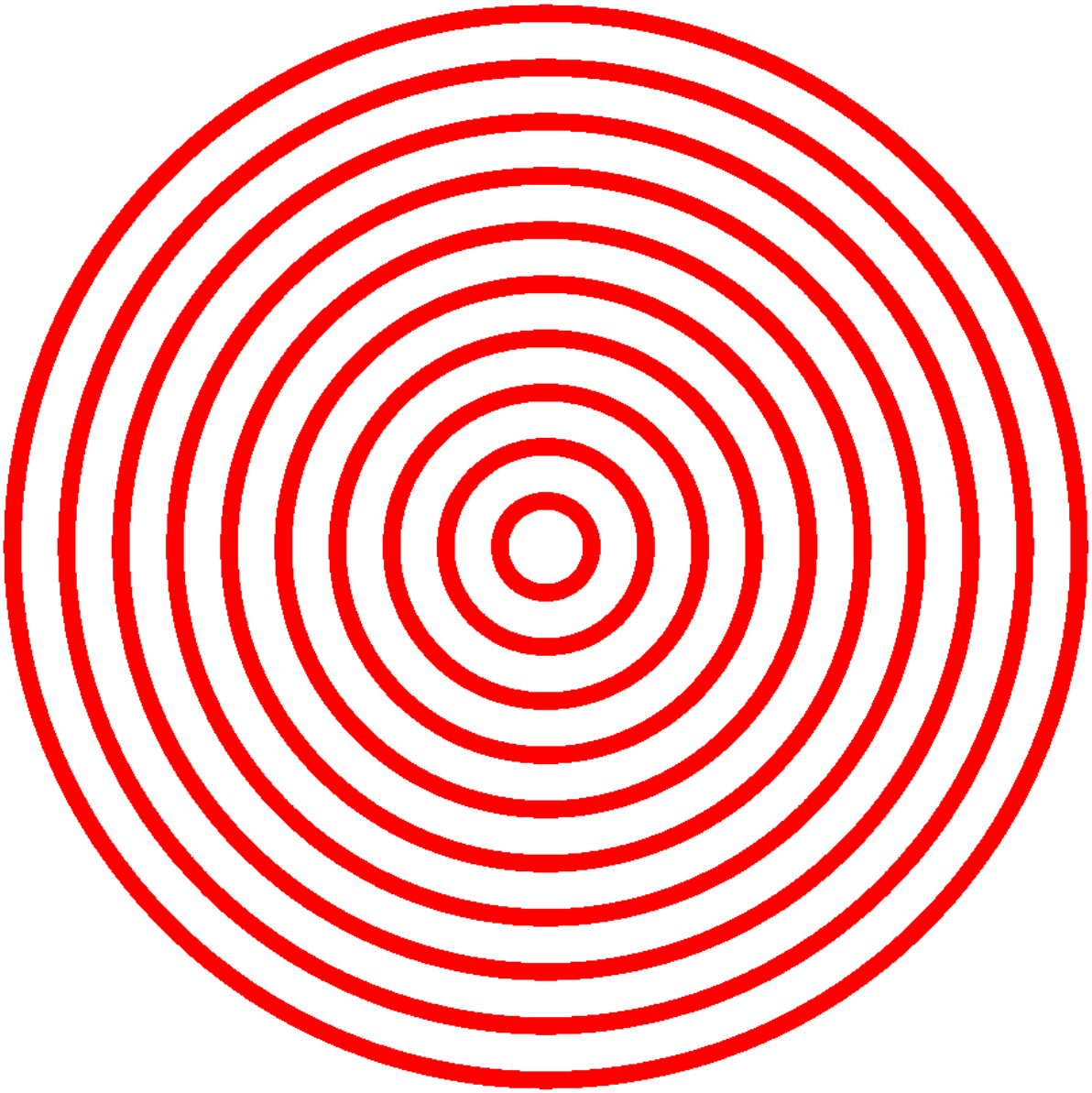
Args:

- **diameter** (float): Total diameter of the target marker
- **ring_width** (float): Width of each ring

Keyword Args:

- **num_rings** (float): Number of concentric rings in the target. Defaults to 10

- **center** (tuple): Coordinate (x1, y1) of the center of the cross. Defaults to (0,0)
- **layer** (int): Layer to place the marker on. Defaults to 1
- **datatype** (int): Datatype to place the marker on. Defaults to 0



Metal Templates, Routes, & Bondpads

Metal Template

class picwriter.components.**MetalTemplate**(*bend_radius=0*, *width=20.0*, *clad_width=20.0*, *resist=u'+'*, *fab=u'ETCH'*, *metal_layer=11*, *metal_datatype=0*, *clad_layer=12*, *clad_datatype=0*)

Template for electrical wires that contains some standard information about the fabrication process and metal wire.

Keyword Args:

- **bend_radius** (float): Radius of curvature for bends in the metal route. Defaults to zero.
- **width** (float): Width of the metal route as shown on the mask. Defaults to 20.
- **clad_width** (float): Width of the cladding (region next to route, mainly used for positive-type photoresists + etching, or negative-type and liftoff). Defaults to 20.
- **resist** (string): Must be either '+' or '-'. Specifies the type of photoresist used. Defaults to *'+'*.
- **fab** (string): If 'ETCH', then keeps resist as is, otherwise changes it from '+' to '-' (or vice versa). This is mainly used to reverse the type of mask used if the fabrication type is 'LIFTOFF'. Defaults to *'ETCH'*.
- **metal_layer** (int): Layer type used for metal route. Defaults to 11.
- **metal_datatype** (int): Data type used for metal route. Defaults to 0.
- **clad_layer** (int): Layer type used for cladding. Defaults to 12.
- **clad_datatype** (int): Data type used for cladding. Defaults to 0.

Metal Routes

class picwriter.components.**MetalRoute**(*trace*, *mt*)

Standard MetalRoute Cell class.

Args:

- **trace** (list): List of coordinates used to generate the route (such as '[(x1,y1), (x2,y2), . . .]'). For now, all trace points must specify 90 degree turns.
- **mt** (MetalTemplate): MetalTemplate object

Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['input'] = {'port': (x1,y1), 'direction': 'dir1'}
- portlist['output'] = {'port': (x2, y2), 'direction': 'dir2'}

Where in the above (x1,y1) are the first elements of 'trace', (x2, y2) are the last elements of 'trace', and 'dir1', 'dir2' are of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*.

Bondpads

class picwriter.components.**Bondpad**(*mt*, *length=150*, *width=100*, *port=(0, 0)*, *direction=u'EAST'*)

Standard Bondpad Cell class.

Args:

- **mt** (MetalTemplate): WaveguideTemplate object

Keyword Args:

- **length** (float): Length of the bondpad. Defaults to 150
- **width** (float): Width of the bondpad. Defaults to 100
- **port** (tuple): Cartesian coordinate of the input port. Defaults to (0,0).
- **direction** (string): Direction that the taper will point *towards*, must be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*. Defaults to *'EAST'*.

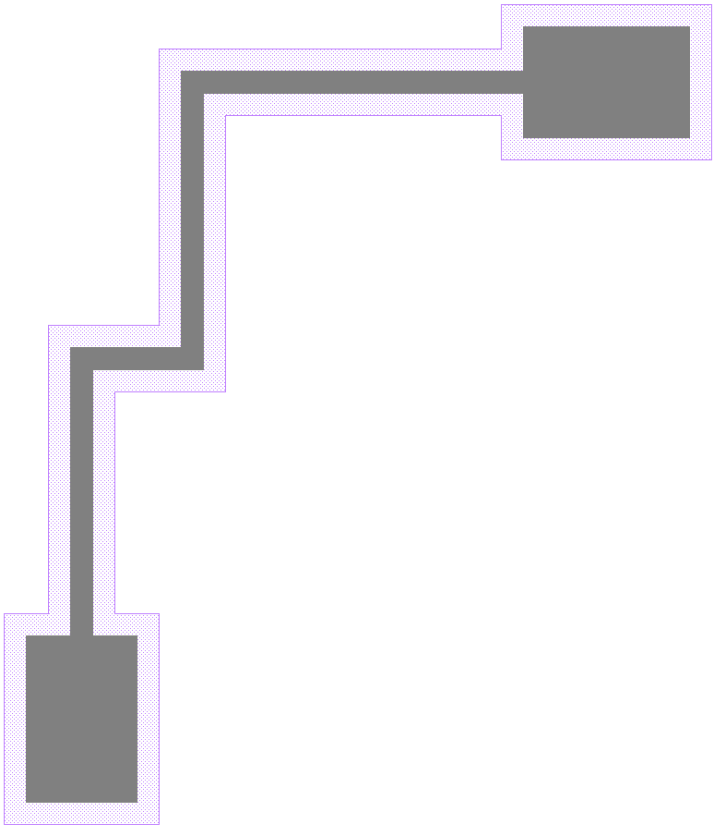
Members:

- **portlist** (dict): Dictionary with the relevant port information

Portlist format:

- portlist['output'] = {'port': (x1, y1), 'direction': 'dir'}

Where in the above (x1,y1) is the same as the 'port' input, and 'dir' is the same as 'direction' input of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*.



The above was generated with:

```
top = gdspy.Cell("top")
mt = MetalTemplate(bend_radius=0, resist='+', fab="ETCH")

mt1=MetalRoute([(0,0), (0,250), (100,250), (100,500), (400,500)], mt)
bp1 = Bondpad(mt, **mt1.portlist["output"])
```

(continues on next page)

(continued from previous page)

```
bp2 = Bondpad(mt, **mt1.portlist["input"])
tk.add(top, bp1)
tk.add(top, bp2)
tk.add(top, mt1)
```

2.1.4 Simulations with PICwriter

The 3-dimensional geometry of all PICwriter components (a scalar dielectric function) can be exported to an HDF5 file format for simulation using various open-source and commercial electromagnetic simulation software. Convenient functions are also provided for automatically computing the transmission/reflection spectra of PICwriter components with arbitrary numbers of ports (specified by their portlist) using freely available finite-difference time-domain (FDTD) software (MEEP).

In order to specify the vertical profile in the vertical (out-of-plane) direction, the user must first specify a **'MaterialStack'** object that is responsible for mapping each mask layer/datatype to a vertical stack of materials.

Defining 3D geometries using PICwriter

The mapping of GDSII layers/datatypes to vertical dielectric profiles is done by creating a *MaterialStack* object and adding *VStacks* for each layer/datatype to be considered. An example of creating a MaterialStack is given below:

```
import picwriter.picsim as ps

epsSiO2 = 1.444**2
epsSi = 3.55**2
etch_stack = [(epsSiO2, 1.89), (epsSi, 0.07), (epsSiO2, 2.04)]
mstack = ps.MaterialStack(vsize=4.0, default_stack=etch_stack, name="Si waveguide")
waveguide_stack = [(epsSiO2, 1.89), (epsSi, 0.22), (epsSiO2, 1.89)]
clad_stack = [(epsSiO2, 1.89), (epsSi, 0.05), (epsSiO2, 2.06)]
mstack.addVStack(layer=1, datatype=0, stack=waveguide_stack)
mstack.addVStack(layer=2, datatype=0, stack=clad_stack)
```

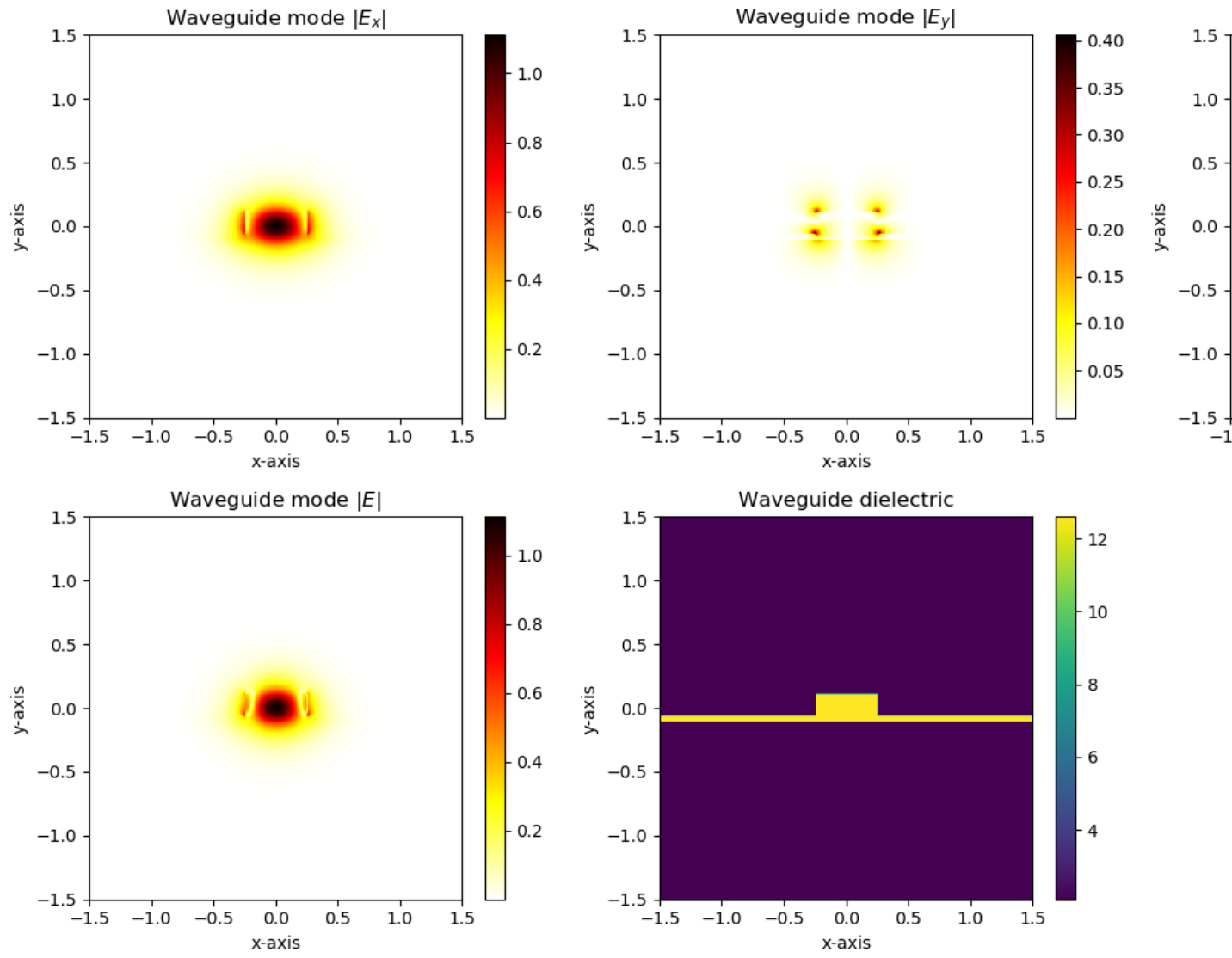
First, we import the picsim library Next, we specified the dielectric constant for the two materials considered (silicon and silicon dioxide at a wavelength of 1550 nm). Then, we create a VStack list (*etch_stack*), that is in the format [(dielectric1, thickness1), (dielectric2, thickness2), . . .]. Withthis we can create the MaterialStack object, with *etch_stack* the default vertical stack in the domain. Next, we create a *waveguide_stack* VStack list and associate it with the (1,0) GDSII layer using the *addVStack* call.

Quickly computing mode profiles

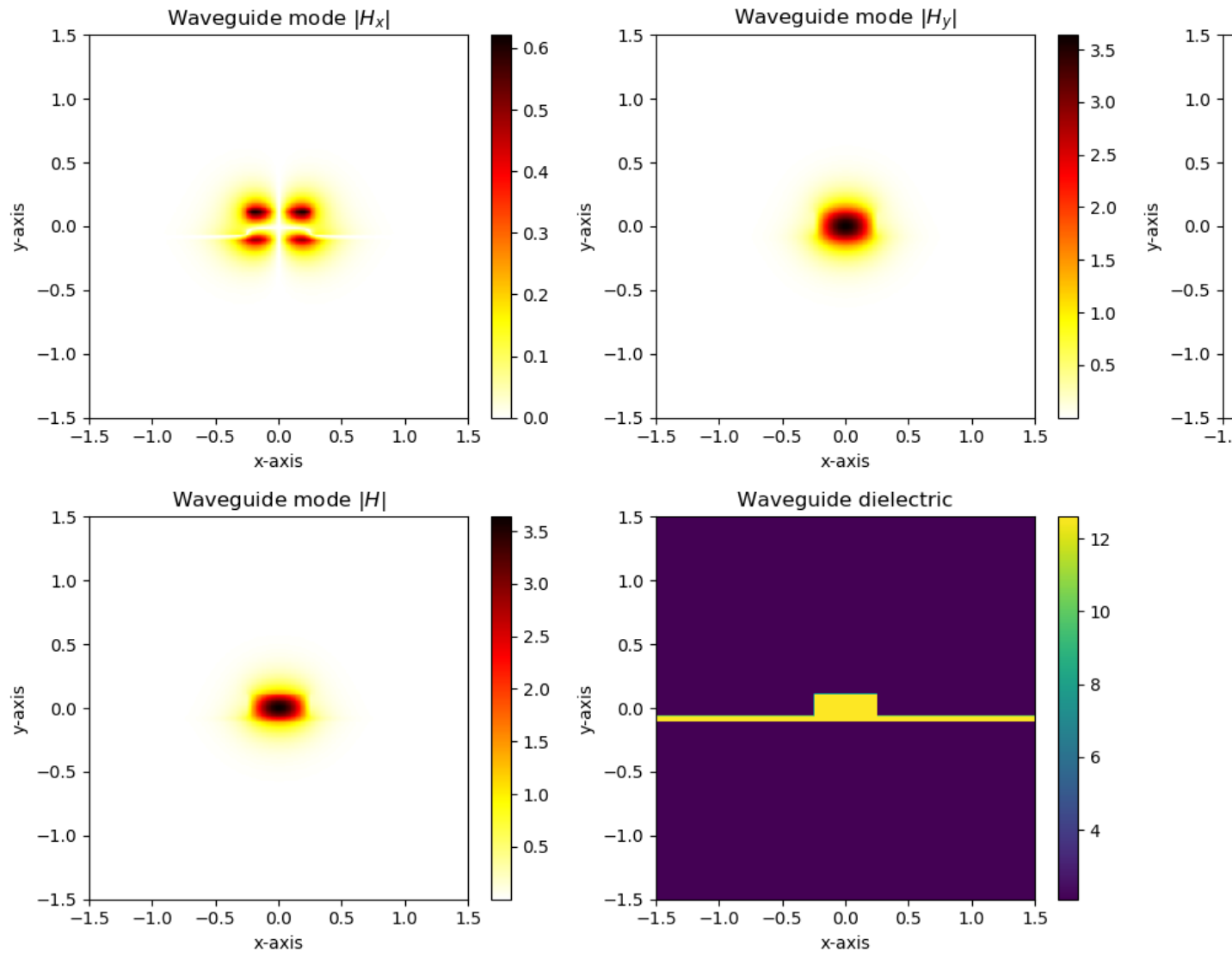
With a properly defined material stack, PICsim makes it easy to quickly view the mode profile corresponding to your layout's WaveguideTemplate with the *compute_mode()* function:

```
import picwriter.components as pc

wgt = pc.WaveguideTemplate(bend_radius=15, wg_width=0.5, clad_width=3.0,
                           wg_layer=1, wg_datatype=0, clad_layer=2, clad_datatype=0)

ps.compute_mode(wgt, mstack, res=128, wavelength=1.55, sx=3.0, sy=3.0,
                plot_mode_number=1, polarization="TE")
```

Which produces plots of the corresponding electric fields:



and magnetic fields:



Computing the transmission/reflection spectra

Likewise, we can build a PICwriter component in the normal way and directly launch a MEEP simulation. Below we build a DirectionalCoupler object and give it 2 um of waveguide at all the inputs/outputs (this will be useful when we simulate with MEEP later):

```
import picwriter.toolkit as tk
from picwriter.components import *
import numpy as np
import gdspy

top = gdspy.Cell("top")
wgt = WaveguideTemplate(bend_radius=15, wg_width=0.5, clad_width=3.0,
                        wg_layer=1, wg_datatype=0, clad_layer=2, clad_datatype=0)

simulated_component = gdspy.Cell('sc')
```

(continues on next page)

(continued from previous page)

```
dc = DirectionalCoupler(wgt, 3.5, 0.2, angle=np.pi/16.0,
                                              parity=1, direction='EAST', port=(0,0))
tk.add(simulated_component, dc)

x0,y0 = dc.portlist['input_top']['port']
x1,y1 = dc.portlist['input_bot']['port']
x2,y2 = dc.portlist['output_top']['port']
x3,y3 = dc.portlist['output_bot']['port']

PML_wg1 = Waveguide([(x0,y0), (x0-2,y0)], wgt)
PML_wg2 = Waveguide([(x1,y1), (x1-2,y1)], wgt)
PML_wg3 = Waveguide([(x2,y2), (x2+2,y2)], wgt)
PML_wg4 = Waveguide([(x3,y3), (x3+2,y3)], wgt)

tk.add(simulated_component, PML_wg1)
tk.add(simulated_component, PML_wg2)
tk.add(simulated_component, PML_wg3)
tk.add(simulated_component, PML_wg4)
```

The gdspy Cell object *simulated_component* now contains four short waveguides and a DirectionalCoupler object. In order to launch a MEEP simulation and compute transmission/reflection spectra, we need to tell PICwriter what *ports* we want to monitor the flux through:

```
ports = [dc.portlist['input_top'],
         dc.portlist['input_bot'],
         dc.portlist['output_top'],
         dc.portlist['output_bot']]
```

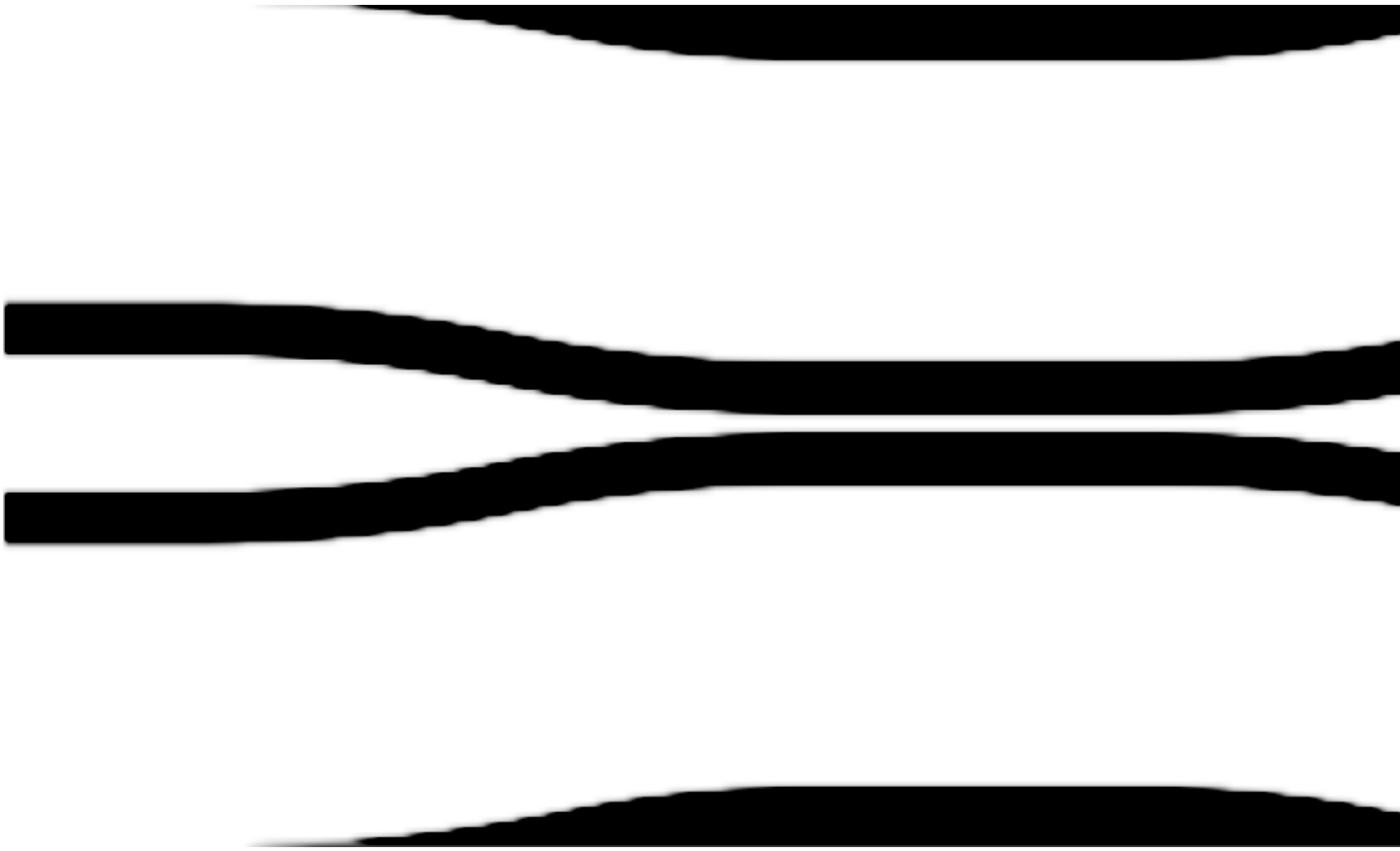
The first port specified in the list above will be the inport where MEEP will place an EigenmodeSource. The last step is calling *compute_transmission_spectra* with the simulated component, MaterialStack, ports, and some additional information about the simulation:

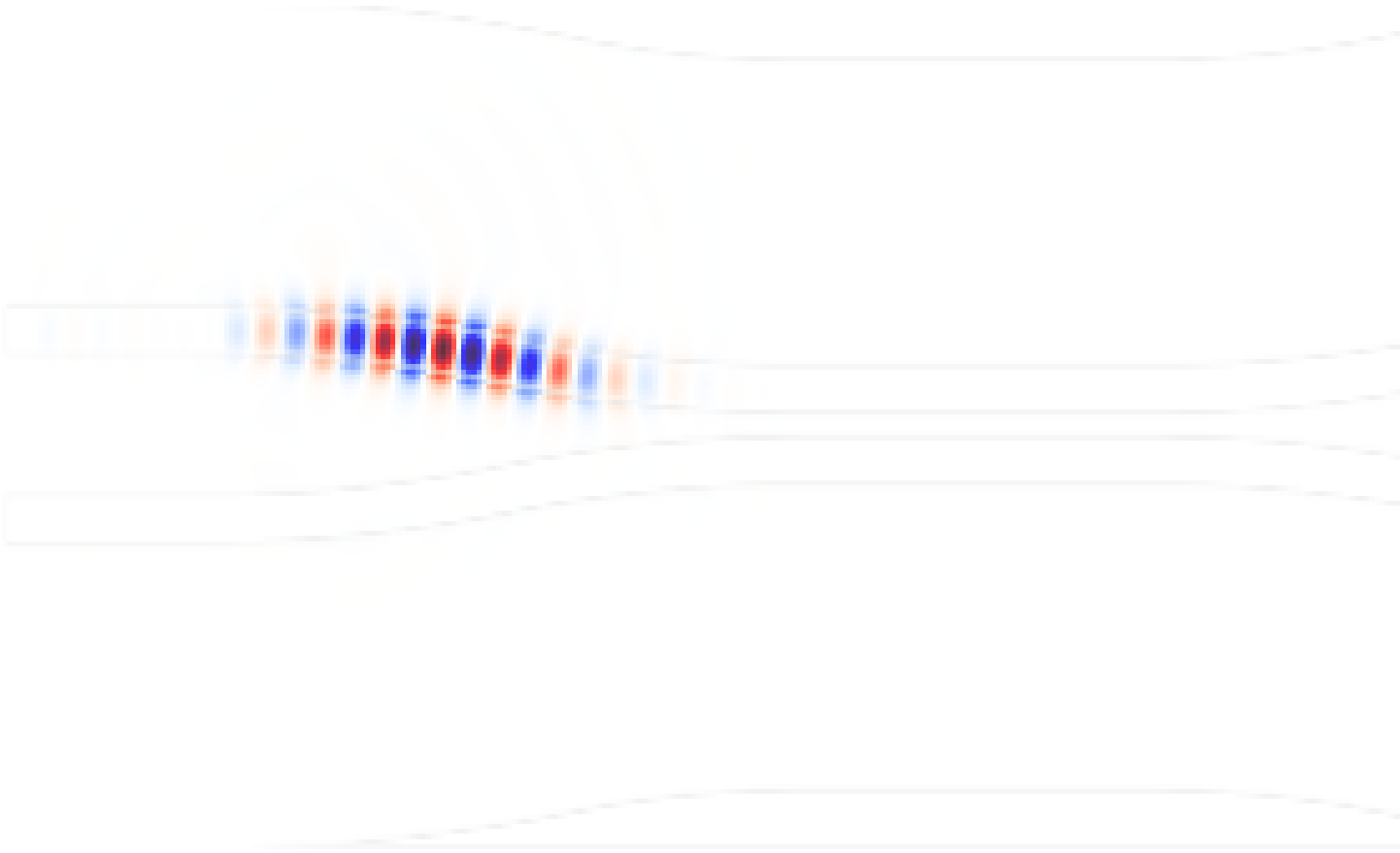
```
ps.compute_transmission_spectra(simulated_component, mstack, wgt, ports, port_
↪vcenter=0,
                                port_height=1.5*0.22, port_width=1.5*wgt.wg_width, dpml=0.
↪5,
                                res=20, wl_center=1.55, wl_span=0.6, fields=True,
                                norm=True, parallel=True, n_p=4)
```

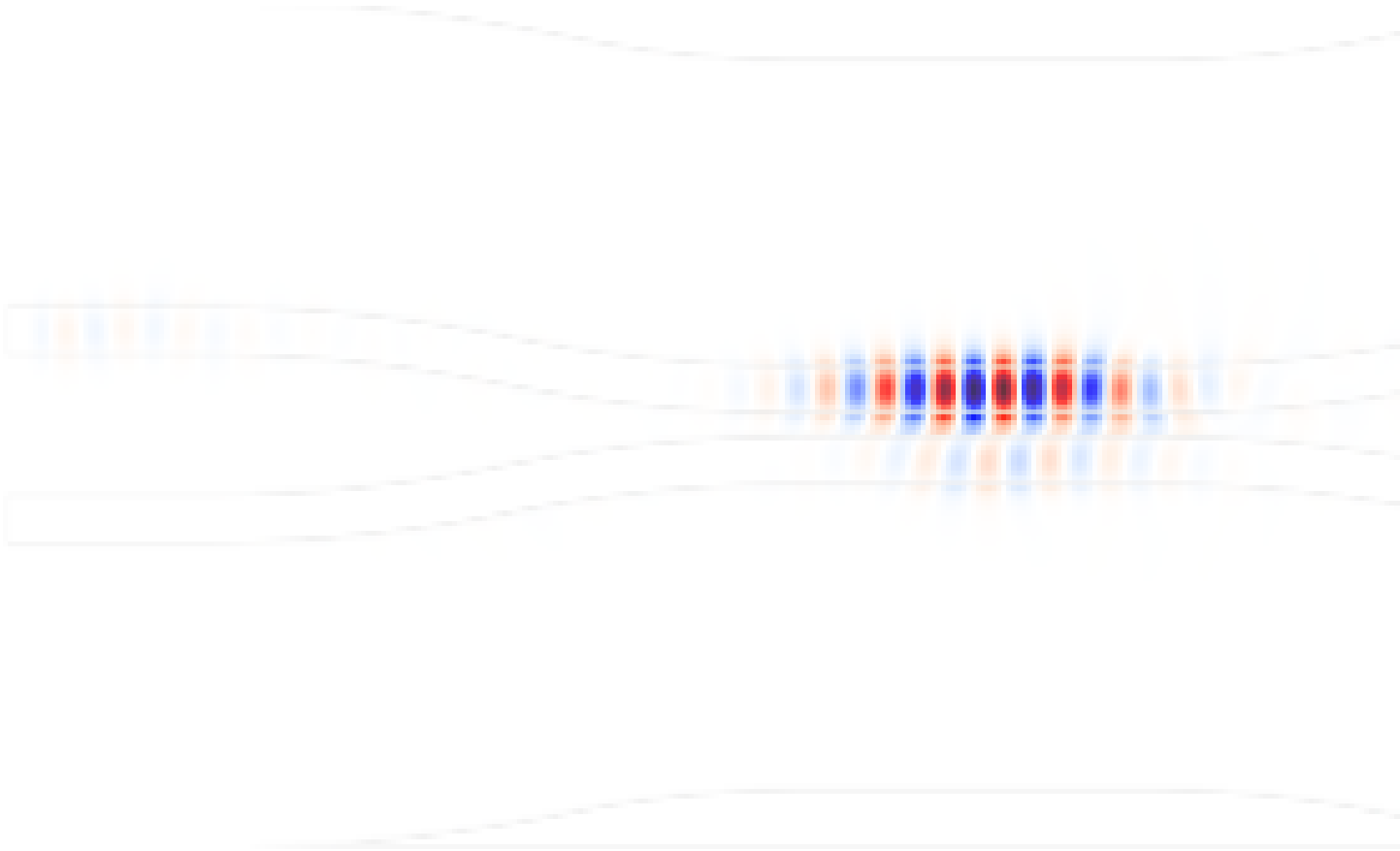
In the above *port_vcenter* specifies the center of the port in the vertical direction, *port_height* and *port_width* are the cross-sectional size of the power flux planes, *res* is the resolution (in pixels/um), *wl_center* and *wl_span* specify the center wavelength and wavelength span of the input pulse (in um), *fields* = True tells MEEP to output images of the electric field profile every few timesteps, *norm* = True tells MEEP to first perform a normalization calculation (straight waveguide) using the *wgt* WaveguideTemplate parameters. *parallel* specifies if the simulation should be run using multiple processor cores (requires MEEP/MPB to be built using parallel libaries), and *n_p* then specifies the number of cores to run on.

NOTE: This function requires MEEP and MPB to be compiled (from source) together, so that MEEP can call MPB to input an EigenmodeSource at the first port location.

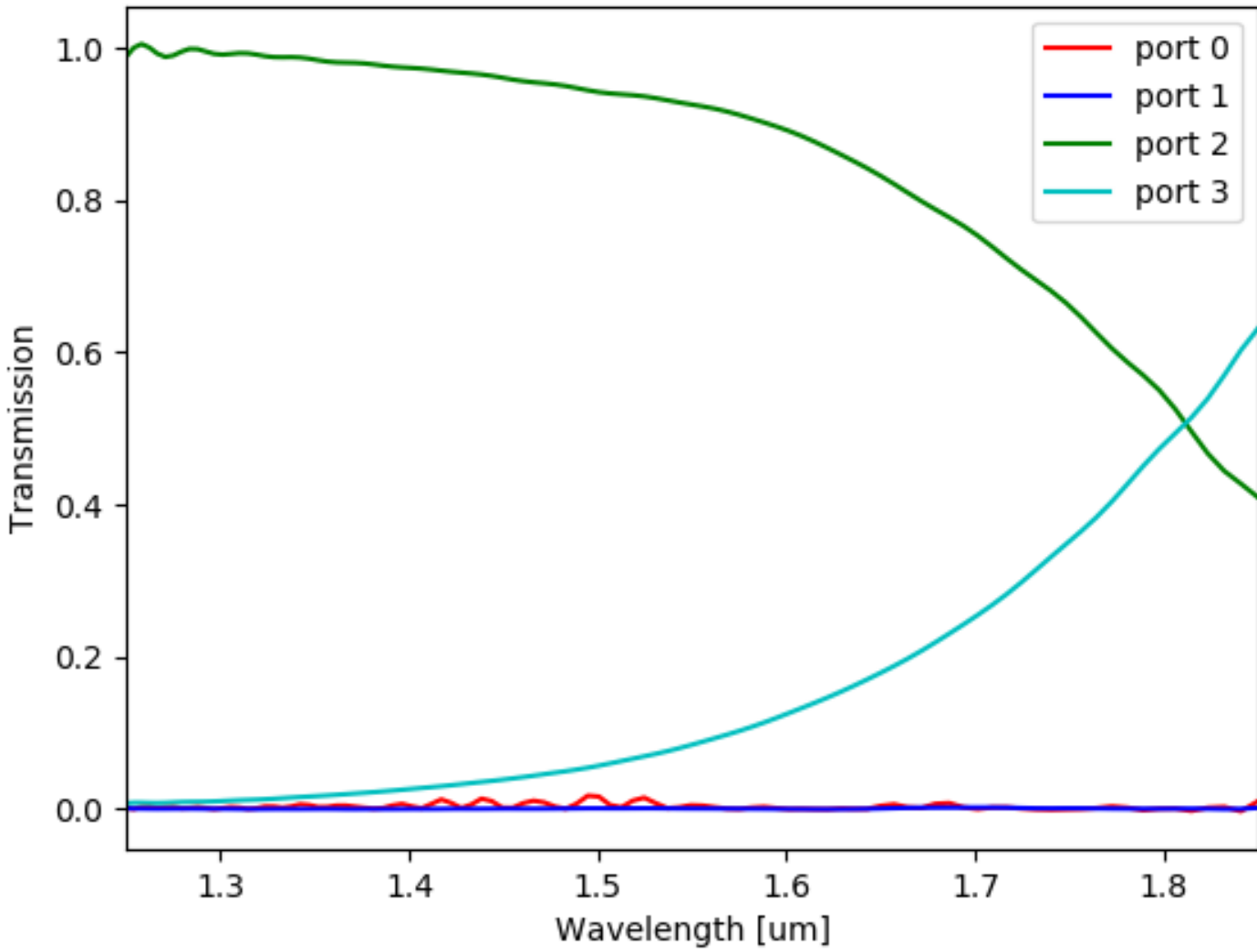
The resulting structure that is simulated and several field images are shown below:







The *compute_transmission_spectra()* function will also compute and plot the appropriately normalized transmission/reflection spectra, saving the .png image in the working directory. The raw power flux data is also saved in .out and .dat files in the working directory.



PICsim Documentation

Set of useful functions for converting PICwriter objects from polygons to hdf5 epsilon files that can be easily imported to MEEP or MPB for quick simulations. Functions for launching siulations using MEEP/MPB are also included.

class picwriter.picsim.**MaterialStack**(*vsize*, *default_stack*, *name=u'mstack'*)

Standard template for generating a material stack

Args:

- **vsize** (float): Vertical size of the material stack in microns (um)
- **default_layer** (list): Default VStack with the following format: [(eps1, t1), (eps2, t2), (eps3, t3), . . .] where eps1, eps2, .. are the permittivity (float), and t1, t2, .. are the thicknesses (float) from bottom to top. Note: t1+t2+. . . *must* add up to vsize.

Members:

- **stacklist** (dictionary): Each entry of the stacklist dictionary contains a VStack list.

Keyword Args:

- **name** (string): Identifier (optional) for the material stack

addVStack(*layer*, *datatype*, *stack*)
Adds a vertical layer to the material stack LIST

Args:

- **layer** (int): Layer of the VStack
- **datatype** (int): Datatype of the VStack
- **stack** (list): Vstack list with the following format: [(eps1, t1), (eps2, t2), (eps3, t3), ...] where eps1, eps2, .. are the permittivity (float), and t1, t2, .. are the thicknesses (float) from bottom to top. Note: if t1+t2+... must add up to vsize.

default_stack = None
self.stack below contains a DICT of all the VStack lists

get_eps(*key*, *height*)
Returns the dielectric constant (epsilon) corresponding to the *height* and VStack specified by *key*, where the height range is zero centered (-vsize/2.0, +vsize/2.0).

Args:

- **key** (layer,datatype): Key value of the VStack being used
- **height** (float): Vertical position of the desired epsilon value (must be between -vsize/2.0, vsize/2.0)

picwriter.picsim.**compute_mode**(*wgt*, *mstack*, *res*, *wavelength*, *sx*, *sy*, *plot_mode_number=1*, *polarization=u'TE'*, *output_directory=u'mpb-sim'*, *save_mode_data=True*, *suppress_window=False*)
Launches a MPB simulation to quickly compute and visualize a waveguide's electromagnetic eigenmodes

Args:

- **wgt** (WaveguideTemplate): WaveguideTemplate object used to specify the waveguide geometry (mask-level)
- **mstack** (MaterialStack): MaterialStack object that maps the gds layers to a physical stack
- **res** (int): Resolution of the MPB simulation (number of pixels per micron).
- **wavelength** (float): Wavelength in microns.
- **sx** (float): Size of the simulation region in the x-direction.
- **sy** (float): Size of the simulation region in the y-direction.
- **plot_mode_number** (int): Which mode to plot (only plots one mode at a time). Must be a number equal to or less than num_modes. Defaults to 1.
- **polarization** (string): Mode polarization. Must be either "TE", "TM", or "None" (corresponding to MPB parities of ODD-X, EVEN-X, or NO-PARITY).
- **output_directory** (string): Output directory for files generated. Defaults to 'mpb-sim'.
- **save_mode_data** (Boolean): Save the mode image and data to a separate file. Defaults to True.
- **suppress_window** (Boolean): Suppress the matplotlib window. Defaults to false.

Returns: List of values for the modes: [[$n_{eff,1}, n_{g,1}$], [$n_{eff,2}, n_{g,2}$], ...]

picwriter.picsim.**compute_transmission_spectra**(*pic_component*, *mstack*, *wgt*, *ports*, *port_vcenter*, *port_height*, *port_width*, *res*, *wl_center*, *wl_span*, *boolean_operations=None*, *norm=False*, *input_pol=u'TE'*, *nfreq=100*, *dpml=0.5*, *fields=False*, *plot_window=False*, *source_offset=0.1*, *symmetry=None*, *skip_sim=False*, *output_directory=u'meep-sim'*, *parallel=False*, *n_p=2*)

Launches a MEEP simulation to compute the transmission/reflection spectra from each of the component's ports when light enters at the input *port*.

How this function maps the GDSII layers to the material stack is something that will be improved in the future. Currently works well for 1 or 2 layer devices. **Currently only supports components with port-directions that are 'EAST' (0) or 'WEST' (pi)**

Args:

- **pic_component** (gdspy.Cell): Cell object (component of the PICwriter library)
- **mstack** (MaterialStack): MaterialStack object that maps the gds layers to a physical stack
- **wgt** (WaveguideTemplate): Waveguide template
- **ports** (list of *Port* dicts): These are the ports to track the Poynting flux through. **IMPORTANT** The first element of this list is where the Eigenmode source will be input.
- **port_vcenter** (float): Vertical center of the waveguide
- **port_height** (float): Height of the port cross-section (flux plane)
- **port_width** (float): Width of the port cross-section (flux plane)
- **res** (int): Resolution of the MEEP simulation
- **wl_center** (float): Center wavelength (in microns)
- **wl_span** (float): Wavelength span (determines the pulse width)

Keyword Args:

- **boolean_operations (list): A list of specified boolean operations to be performed on the layers (ORDER MATTE**
  [((layer1/datatype1), (layer2/datatype2), operation), ...] where 'operation' can be 'xor', 'or', 'and', or 'not' and the resulting polygons are placed on (layer1, datatype1). See below for example.
- **norm** (boolean): If True, first computes a normalization run (transmission through a straight waveguide defined by *wgt* above. Defaults to *False*. If *True*, a WaveguideTemplate must be specified.
- **input_pol** (String): Input polarization of the waveguide mode. Must be either "TE" or "TM". Defaults to "TE" (z-antisymmetric).
- **nfreq** (int): Number of frequencies (wavelengths) to compute the spectrum over. Defaults to 100.
- **dpml** (float): Length (in microns) of the perfectly-matched layer (PML) at simulation boundaries. Defaults to 0.5 um.
- **fields** (boolean): If true, outputs the epsilon and cross-sectional fields. Defaults to false.
- **plot_window** (boolean): If true, outputs the spectrum plot in a matplotlib window (in addition to saving). Defaults to False.
- **source_offset** (float): Offset (in x-direction) between reflection monitor and source. Defaults to 0.1 um.

- **skip_sim** (boolean): Defaults to False. If True, skips the simulation (and hdf5 export). Useful if you forgot to perform a normalization and don't want to redo the whole MEEP simulation.
- **output_directory** (string): Output directory for files generated. Defaults to 'meep-sim'.
- **parallel** (boolean): If *True*, will run simulation on *np* cores (*np* must be specified below, and MEEP/MPB must be built from source with parallel-libraries). Defaults to False.
- **n_p** (int): Number of processors to run meep simulation on. Defaults to *2*.

Example of boolean_operations (using the default):

The following default boolean_operation will:

(1) do an 'xor' of the default layerset (-1,-1) with a cladding (2,0) and then make this the new default layerset

(2) do an 'xor' of the cladding (2,0) and waveguide (1,0) and make this the new cladding

boolean_operations = [((-1,-1), (2,0), 'and'), ((2,0), (1,0), 'xor')]

picwriter.picsim.**export_component_to_hdf5**(*filename*, *component*, *mstack*, *boolean_operations*)

Outputs the polygons corresponding to the desired component and MaterialStack. Format is compatible for generating prism geometries in MEEP/MPB.

Note: that the top-down view of the device is the 'X-Z' plane. The 'Y' direction specifies the vertical height.

Args:

- **filename** (string): Filename to save (must end with '.h5')
- **component** (gdspy.Cell): Cell object (component of the PICwriter library)
- **mstack** (MaterialStack): MaterialStack object that maps the gds layers to a physical stack
- **boolean_operations** (list): A list of specified boolean operations to be performed on the layers (order matters, see below).

The boolean_operations argument must be specified in the following format:

```
boolean_opeartions = [((layer1/datatype1), (layer2/datatype2), operation), ...]
```

where 'operation' can be 'xor', 'or', 'and', or 'not' and the resulting polygons are placed on (layer1, datatype1). For example, the boolean_operation below:

```
boolean_operations = [((-1,-1), (2,0), 'and'), ((2,0), (1,0), 'xor')]
```

will:

(1) do an 'xor' of the default layerset (-1,-1) with the cladding (2,0) and then make this the new default

(2) do an 'xor' of the cladding (2,0) and waveguide (1,0) and make this the new cladding

Write format:

- LL = layer
- DD = datatype
- NN = polygon index
- VV = vertex index
- XX = x-position
- ZZ = z-position

- height = height of the prism
- eps = epsilon of the prism
- y-center = center (y-direction) of the prism [note: (x,y) center defaults to (0,0)]

picwriter.picsim.**export_wgt_to_hdf5** (*filename*, *wgt*, *mstack*, *sx*)

Outputs the polygons corresponding to the desired waveguide template and MaterialStack. Format is compatible for generating prism geometries in MEEP/MPB.

Note: that the top-down view of the device is the 'X-Z' plane. The 'Y' direction specifies the vertical height.

Args:

- **filename** (string): Filename to save (must end with '.h5')
- **wgt** (WaveguideTemplate): WaveguideTemplate object from the PICwriter library
- **mstack** (MaterialStack): MaterialStack object that maps the gds layers to a physical stack
- **sx** (float): Size of the simulation region in the x-direction

Write-format for all blocks:

- CX = center-x
- CY = center-y
- width = width (x-direction) of block
- height = height (y-direction) of block
- eps = dielectric constant

2.1.5 Toolkit Documentation

Set of helper functions that make it easier to manipulate and work with gdspy subclasses defined in **components** module

class picwriter.toolkit.**Component** (*name*, **args*)

Super class for all objects created in PICwriter. This class handles rotations, naming, etc. for all components, so that writing python code for new cells requires less overhead. Component is a wrapper around gdspy Cell objects.

Args:

- **name** (string): The name prefix to be used for these

Keyword Args:

- **angle** (float): Angle in radians (between 0 and pi/2) at which the waveguide bends towards the coupling region. Default=pi/6.

add (*element*, *origin=(0, 0)*, *rotation=0.0*, *x_reflection=False*)

Add a reference to an element or list of elements to the cell associated with this component

picwriter.toolkit.**add** (*top_cell*, *component_cell*, *center=(0, 0)*, *x_reflection=False*)

First creates a CellReference to subcell, then adds this to topcell at location center.

Args:

- **top_cell** (gdspy.Cell): Cell being added to
- **component_cell** (gdspy.Cell): Cell of the component being added

Keyword Args:

- **port** (tuple): location for the subcell to be added
- **direction** (string): Direction that the component will point *towards*, can be of type *'NORTH'*, *'WEST'*, *'SOUTH'*, *'EAST'*, OR an angle (float, in radians). Defaults to 'EAST' (zero degrees of rotation).

Returns: None

picwriter.toolkit.**build_mask**(*cell*, *wgt*, *final_layer=None*, *final_datatype=None*)

Builds the appropriate mask according to the resist specifications and fabrication type. Does this by applying a boolean 'XOR' or 'AND' operation on the waveguide and clad masks.

Args:

- **cell** (gdspy.Cell): Cell with components. Final mask is placed in this cell.
- **wgt** (WaveguideTemplate): Waveguide template containing the resist information, and layers/datatypes for the waveguides and cladding.

Keyword Args:

- **final_layer** (int): layer to place the mask on (defaults to *wgt.clad_layer + 1*)
- **final_datatype** (int): datatype to place the mask on (defaults to *0*)

Returns: None

picwriter.toolkit.**build_waveguide_polygon**(*func*, *wg_width*, *start_direction*, *end_direction*, *start_val=0*, *end_val=1*, *grid=0.001*)

Args:

- **func** (function): Function that takes a single (floating point) argument, and returns a (x,y) tuple.
- **wg_width** (float): Waveguide width
- **num_pts** (int): Number of points that make up the waveguide path
- **start_direction** (float): Starting direction of the path, in *radians*.
- **end_direction** (float): End direction of the path, in *radians*.

Keyword Args:

- **start_val** (float): Starting value (argument passed to *func*). Defaults to 0.
- **end_val** (float): Ending value (argument passed to *func*). Defaults to 1.
- **grid** (float): Grid resolution used to determine when curve length has converged. Guarantees that polygon formed by the points results in no more than a grid/2.0 error from the true position. Defaults to 0.001

Returns: Two lists, one for each edge of the waveguide.

picwriter.toolkit.**dist**(*pt1*, *pt2*)

Given two cardinal points, returns the distance between the two.

Args:

- **pt1** (tuple): Point 1
- **pt2** (tuple): Point 2

Returns: float Distance

Example:

```
import picwriter.toolkit as tk
print(tk.dist((0, 0), (100, 100)))
```

The above prints 141.42135623730951

picwriter.toolkit.**flip_direction**(*direction*)

Returns the opposite of *direction*, where each direction is either `'NORTH'`, `'WEST'`, `'SOUTH'`, or `'EAST'`

Args:

- **direction** (direction): Direction to be flipped
- **pt2** (tuple): Point 2

Returns: direction (`'NORTH'`, `'WEST'`, `'SOUTH'`, or `'EAST'`)

picwriter.toolkit.**get_angle**(*pt1*, *pt2*)

Given two cardinal points, returns the corresponding angle in *radians*. Must be an integer multiple of pi/2.

Args:

- **pt1** (tuple): Point 1
- **pt2** (tuple): Point 2

Returns: float Angle (integer multiple of pi/2)

Example:

```
import picwriter.toolkit as tk
print(tk.get_angle((0, 0), (0, 100)))
```

The above prints 1.5707963267948966

picwriter.toolkit.**get_curve_length**(*func*, *start*, *end*, *grid=0.001*)

Returns the length (in microns) of a curve defined by the function *func* on the interval [start, end]

Args:

- **func** (function): Function that takes a single (floating point) argument, and returns a (x,y) tuple.
- **start** (float): Starting value (argument passed to *func*).
- **end** (float): Ending value (argument passed to *func*).

Keyword Args:

- **grid** (float): Grid resolution used to determine when curve length has converged. Defaults to 0.001.

Returns: float Length

picwriter.toolkit.**get_direction**(*pt1*, *pt2*)

Returns a cardinal direction (`'NORTH'`, `'WEST'`, `'SOUTH'`, and `'EAST'`) that corresponds to a cartesian point *pt1 (tuple), pointing TOWARDS a second point 'pt2*

Args:

- **pt1** (tuple): Point 1
- **pt2** (tuple): Point 2

Returns: string (`'NORTH'`, `'WEST'`, `'SOUTH'`, and `'EAST'`)

Example:

```
import picwriter.toolkit as tk
tk.get_direction((0,0), (-100,0))
```

The above prints 'WEST'

picwriter.toolkit.**get_exact_angle**(*pt1*, *pt2*)
Given two cardinal points, returns the corresponding angle in *radians*.

Args:

- **pt1** (tuple): Point 1
- **pt2** (tuple): Point 2

Returns: float Angle (in radians)

Example:

```
import picwriter.toolkit as tk
print(tk.get_angle((0, 0), (100, 100)))
```

The above prints 0.785398163

picwriter.toolkit.**get_keys**(*cell*)
Returns a list of the keys available in a portlist, such as 'input', 'output', 'top_output', etc. Only works for picwriter components.

Args:

- **cell** (gdspy.Cell): Cell from which to get get the portlist

Returns: List of portlist keys corresponding to 'cell'.

picwriter.toolkit.**get_trace_length**(*trace*, *wgt*)
Returns the total length of a curved waveguide trace.

Args:

- **trace** (list): tracelist of (x,y) points all specifying 90 degree angles.
- **wgt** (WaveguideTemplate): template for the waveguide, the bend_radius of which is used to compute the length of the curved section.

Returns: float corresponding to the length of the waveguide trace

picwriter.toolkit.**get_turn**(*dir1*, *dir2*)
Returns an angle (+pi/2 or -pi/2) corresponding to the CW or CCW turns that takes you from direction *dir1* to *dir2*, where each direction is either `'NORTH'`, `'WEST'`, `'SOUTH'`, or `'EAST'`

Args:

- **dir1** (direction): Point 1
- **pt2** (tuple): Point 2

Returns: float (+pi/2 or -pi/2)

picwriter.toolkit.**normalize_angle**(*angle*)
Returns the angle (in radians) between -pi and +pi that corresponds to the input angle

Args:

- **angle** (float): Angle to normalize

Returns: float Angle

picwriter.toolkit.**translate_point**(*pt*, *length*, *direction*, *height=0.0*)
Returns the point (tuple) corresponding to *pt* translated by distance *length* in direction *direction* where each direction is either `'NORTH'`, `'WEST'`, `'SOUTH'`, or `'EAST'`

Args:

- **pt** (tuple): Starting point
- **length** (float): Distance to move in *direction*
- **direction** (direction): Direction to move in

Keyword Args:

- **height** (float): Distance to move perpendicular to *direction*. Defaults to 0.

Returns: point, tuple (x, y)

2.1.6 License

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

p

Index

P

R

S

T

W