GDSHelpers Documentation

Release 1.2.1

QPIT Münster

Jan 26, 2022

Contents

The gdshelpers-package is a design framework for lithography-pattern files, developed primarily for integrated optics. It builds on the shoulder of the Shapely (http://toblerity.org/shapely/index.html) project. Please make sure that you have an initial understanding of this. Afterwards this package should be simple to understand.

It includes adapters to convert any Shapely objects to gdsCAD/gdspy objects - including correct fracturing of polygons and lines in elements with less than 200 points. An archaic restriction of the GDII file format. Furthermore, it allows to save the design in the more recent OASIS-format using the fatamorgana-library, which allows smaller files-sizes and has less restrictions. The user can use an abstraction layer above these libraries, which allows convenient exchange of the underlying library and therefore changing the export-format by a parameter.

In addition it includes a growing selection of optical and superconducting parts, including:

- A waveguide part, allowing easy chaining of bends and straight waveguides.
  - Includes parameterized paths and Bézier curves.
  - Automatic smooth connection to a target point/port
  - The size of the waveguide can be tapered (linear or by a user defined function), which can e.g. be used for optical edge coupling or electronic contact pads
  - Allows to design slot-waveguides and coplanar waveguides (with arbitrary number of rails)
- Different types of splitters:
  - Y-splitter
  - MMI-splitters
  - Directional splitter
- Couplers
  - Grating couplers (allowing apodized gratings)
  - Tapers for hybrid 3D-integration
- Ring and racetrack resonators
- Mach-Zehnder interferometers
- Spirals
- Superconducting nanowire single photon detectors (SNSPDs)
- Superconducting nanoscale Transistors (NTRONs)
- Different types of markers
- QRcodes
- A possibility to include images
- Text-elements for labeling the structures
- GDSII-import

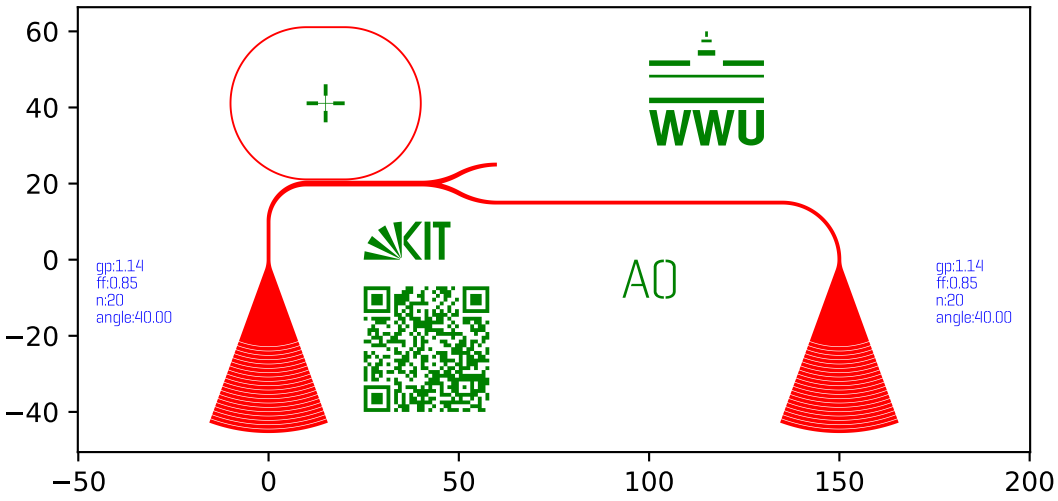
The library itself is designed to be as simple to use as possible:

```
import numpy as np

from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler
from gdshelpers.parts.resonator import RingResonator
from gdshelpers.parts.splitter import Splitter
```

(continues on next page)

(continued from previous page)

```
from gdshelpers.parts.logo import KITLogo, WWULogo
from gdshelpers.parts.optical_codes import QRCode
from gdshelpers.parts.text import Text
from gdshelpers.parts.marker import CrossMarker

# Generate a coupler with parameters from the coupler database
coupler1 = GratingCoupler.make_traditional_coupler_from_database([0, 0], 1, 'sn330',␣
↪1550)
coupler2 = GratingCoupler.make_traditional_coupler_from_database([150, 0], 1, 'sn330',
↪ 1550)

coupler1_desc = coupler1.get_description_text(side='left')
coupler2_desc = coupler2.get_description_text(side='right')

# And add a simple waveguide to it
wg1 = Waveguide.make_at_port(coupler1.port)
wg1.add_straight_segment(10)
wg1.add_bend(-np.pi/2, 10, final_width=1.5)

res = RingResonator.make_at_port(wg1.current_port, gap=0.1, radius=20,
                                 race_length=10, res_wg_width=0.5)

wg2 = Waveguide.make_at_port(res.port)
wg2.add_straight_segment(30)
splitter = Splitter.make_at_root_port(wg2.current_port, total_length=20, sep=10, wg_
↪width_branches=1.0)

wg3 = Waveguide.make_at_port(splitter.right_branch_port)
wg3.add_route_single_circle_to_port(coupler2.port)

# Add a marker just for fun
marker = CrossMarker.make_traditional_paddle_markers(res.center_coordinates)

# The fancy stuff
kit_logo = KITLogo([25, 0], 10)
wwu_logo = WWULogo([100, 30], 30, 2)
qr_code = QRCode([25, -40], 'https://www.uni-muenster.de/Physik.PI/Pernice', 1.0)
dev_label = Text([100, 0], 10, 'A0', alignment='center-top')

# Create a Cell to hold the objects
cell = Cell('EXAMPLE')

# Convert parts to gdsCAD polygons
cell.add_to_layer(1, coupler1, wg1, res, wg2, splitter, wg3, coupler2)
cell.add_to_layer(2, wwu_logo, kit_logo, qr_code, dev_label)
cell.add_to_layer(2, marker)
cell.add_to_layer(3, coupler1_desc, coupler2_desc)
cell.show()
```


60
40
20
0
−20
−40
−50
0
50
100
150
200
WWU
KIT
A0
gp:1.14
ff:0.85
n:20
angle:40.00
gp:1.14
ff:0.85
n:20
angle:40.00

CHAPTER 1

Citing GDSHelpers

We would appreciate if you cite the following paper in your publications for which you used GDSHelpers:

Helge Gehring, Matthias Blaicher, Wladick Hartmann, and Wolfram H. P. Pernice, "Python based open source design framework for integrated nanophotonic and superconducting circuitry with 2D-3D-hybrid integration" (https://www.osapublishing.org/osac/abstract.cfm?uri=osac-2-11-3091) OSA Continuum 2, 3091-3101 (2019)

CHAPTER 2

Support

If you have problems using GDSHelpers don't hesitate to contact us using Discussions (https://github.com/HelgeGehring/gdshelpers/discussions) or send me a mail (https://github.com/HelgeGehring)

CHAPTER 3

Table of contents

3.1 Installing a Python GDSII development environment under Windows

3.1.1 Download and install Git

All development of gdshelper is done it Git. Even if you do not plan on contributing by your own, you will eventually need it. Download it from git-scm.com (http://git-scm.com/downloads). Keep the default settings.

3.1.2 Download and install Python

CPython as Python distribution

While you can basically use any Python 3 interpreter (at least Python 3.5, preferably the newest version), in this guide we use CPython by python.org. Head there now and download (https://www.python.org/downloads/) the Windows installer. Make sure that the installer **adds CPython to PATH** and installs the Python package managment **pip**.

PyCharm as IDE

One of the best Python IDEs is PyCharm which now also has a free community edition. Go and get it (http://www.jetbrains.com/pycharm/). It's recommendable to install it via te Jetbrains toolbox (https://www.jetbrains.com/toolbox/), as it simplifies upgrading Pycharm.

You can directly start it after it is installed. On the first start it will ask you about your default theme and keymap. Change it to your own preference.

Setting up Python in PyCharm

Before doing any real work you will have to tell PyCharm which Python it should use. On the welcome screen, select `Configure`, then `Settings`. Add the Python interpreter in `Project Interpreters` and click on the gear in the upper right and then `Add`. There you can add the Python Interpreter by selecting `System Interpreter`. The right path should be already in the form.

PyCharm will then parse all the installed modules of that Python installation.

Installing the gdshelpers and optional dependencies

Even under Windows, the command line is sometimes useful. In our case we use *pip* to install gdshelpers with image-export directly by using the single command:

```
pip install gdshelpers[image_export]
```

For most users this configuration should be sufficient.

Alternatively, you can also add the gdshelpers to your project using `VCS --> Checkout from Version Control --> Git` (the link is https://github.com/HelgeGehring/gdshelpers.git) and then `Attach` to add it to your project. Using this way, it is also possible to modify the gdshelpers and to contribute to the development.

Additionally you need extra packages for certain functions of the package. For exporting the design to the OASIS-format you should install the library *fatamorgana* using `pip install fatamorgana`. In order to create GDSII-files, you can use the included GDSII-export or decide between *gdspy* (fully python 3 compatible, `pip install gdspy`) and *gdsCAD* (also working under python 3, but not installable using *pip*). For directly generating pictures from the designs the package *descartes* needs to be installed (`pip install descartes`).

Updating gdshelpers

In order the update to the most recent version of gdshelpers you should execute from time to time:

```
pip install --upgrade gdshelpers
```

Finish

That's it, you are all set to generate your own GDS file. Head over to the tutorial.

3.2 Tutorial

3.2.1 Introduction

Designing nano-optical devices is a huge and complex task. A lot of steps have to be performed just right. Starting from the initial design to fabrication and testing. This software library is designed to make your life easier in one regard: The *actual* design of the chip.

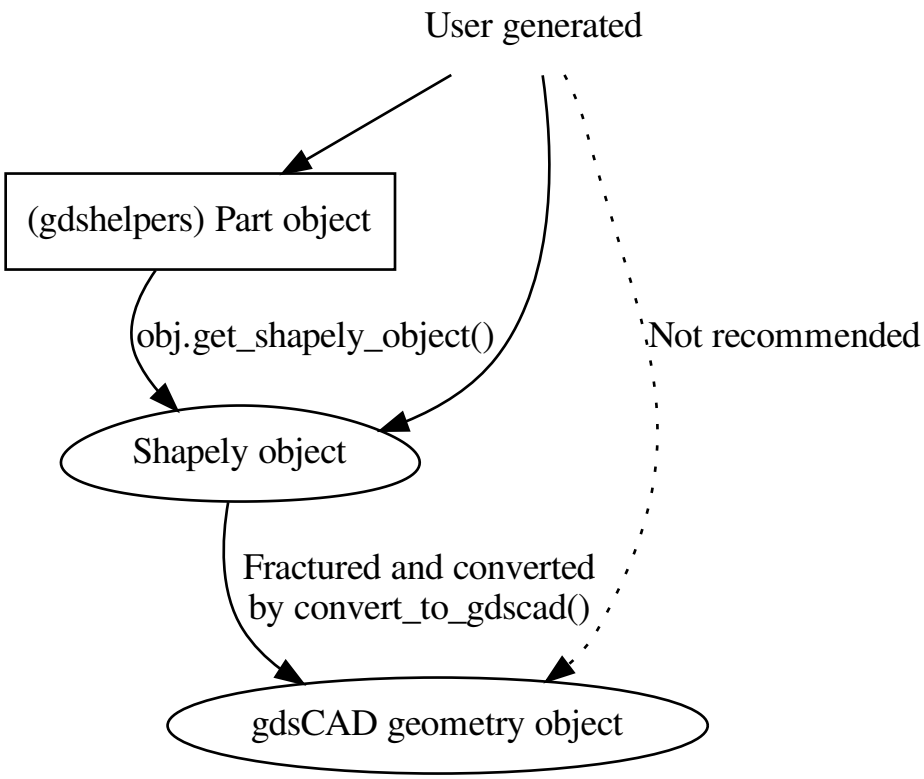
In this tutorial we will create:

- *A simple chip*: First create a simple loop
- *Simple Drawing*: A gentle introduction on how to generate polygons and do boolean operations.

3.2.2 Software design and intended usage

Let's first talk about some fundamental library design choices. If you are new, you might not understand everything. That's OK, though. Head back here once you got the hang of it.

Under normal circumstances all geometric shapes and operations should be generated in Shapely (http://toblerity.org/shapely/manual.html). This will already give you a full repertoire of powerful geometric operations. These objects can then be converted to gdsCAD (http://pythonhosted.org/gdsCAD/) geometric objects. Since those already represent the polygons which will end in the final GDSII (https://en.wikipedia.org/wiki/GDSII) file it also follows the serious restrictions of the GDSII file format. Those restrictions will already be respected when converting the Shapely (http://toblerity.org/shapely/manual.html) object. The conversion function is called *gdshelpers.geometry.convert_to_gdscad()*.



While it looks like you'd need to call `get_shapely_object()` every time you want to convert it to gdsCAD by *convert_to_gdscad()*, you actually do not. If an object passed to *convert_to_gdscad()* has the `get_shapely_object()` method, it will be called for you and the resulting Shapely object will be converted instead. Note, that Shapely objects do not know layers, GDSII datatypes etc. You can specify those when converting to gdsCAD.

3.2.3 A simple chip

Creating the project

In PyCharm, create a new project called `gds_tutorial` at a location of your choice. Make sure you select the virtual environment interpreter which you set up in the install guide.

Add a new python file via `New->New->Python file`, called `chip.py`. An editor opens with a nearly empty file, you may ignore the `__author__` line and delete it at will.

Hello World

To test that everything is working, let's just put a `print "Hello World"` in `chip.py`. Of course this will just print `Hello World` on the console. Right click on `chip.py` and select `Run `chip``.

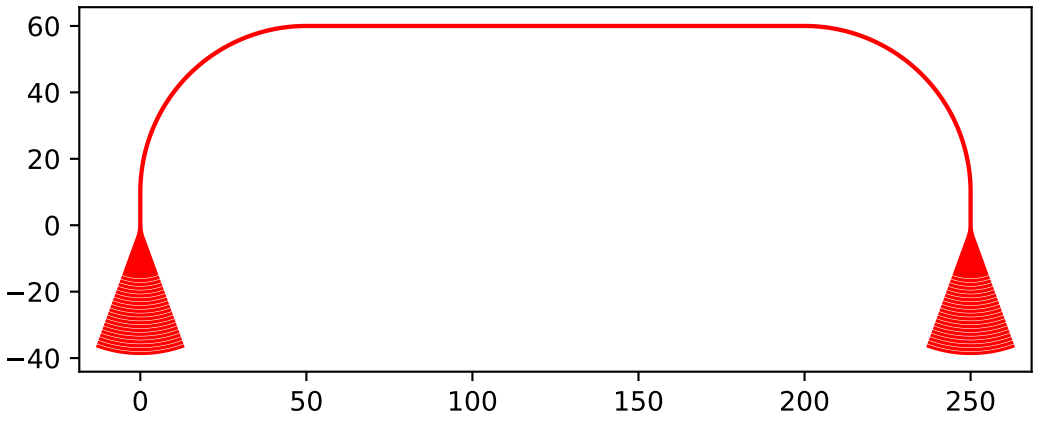
Your program is directly executed and you can see the result on the bottom of the screen in the terminal.

This action has create a run profile for this file. If you create several run profiles, you can switch between them in the top toolbar directly left of the play and the bug button. (Some installations somehow don't show the toolbar by default, if so I recommend enabling it.) The bug button directly starts into the debugger and is an enormous help if you try to find an error in your program.

A first device

Our first device is going to be two grating couplers connected via a waveguide. This will be really simple

```
import numpy as np
from math import pi
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler

coupler_params = {
    'width': 1.3,
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

left_coupler = GratingCoupler.make_traditional_coupler(origin=(0, 0), **coupler_
↪params)
wg = Waveguide.make_at_port(port=left_coupler.port)
wg.add_straight_segment(length=10)
wg.add_bend(angle=-pi / 2, radius=50)
wg.add_straight_segment(length=150)
wg.add_bend(angle=-pi / 2, radius=50)
wg.add_straight_segment(length=10)
right_coupler = GratingCoupler.make_traditional_coupler_at_port(port=wg.current_port,␣
↪**coupler_params)

cell = Cell('SIMPLE_DEVICE')
cell.add_to_layer(1, left_coupler, wg, right_coupler)
cell.show()
# cell.save('chip.gds')
```



Let's go through that step by step:

The imports

The first paragraph contains import statements. These tell python which packages it should now in this program. While the `import` statement just imports the whole package path, the `from ... import ...` statement imports an object to the local namespace. So instead of writing `math.pi` all the time, `from math import pi` allows us to just use `pi` since Python now knows where the `pi` object came from. Several modules are listed here:

- `math` which is part of the Python standard library and also contains stuff such as `sin()` etc.
- `gdshelpers` which is what this tutorial is primarily about.

The part objects

We use two parts here: *`gdshelpers.parts.coupler.GratingCoupler`* and *`gdshelpers.parts.waveguide.Waveguide`* follow the links to get more information on them. When you look again at the source code creating the parts, you will see a `Port` mentioned. This port is just a construct designed to help the user. It bundles three properties inherent to any waveguide:

- Position
- Angle
- Width of the waveguide/port

All parts can also be placed by hand without the usage of ports – but its much simpler to use them.

Output to GDS

We previously created our part objects (`left_coupler`, `wg` and `right_coupler`) but we need to add it to our GDS file somehow. A bit of background might be in order here: GDS files are really really old file formats. They have quite a lot of restrictions – the most serious of them is the limit of 200 points per line or polygon. The device we have just created has definitely more points, so it has to be sliced or 'fractured'. But fear not, the `gdshelpers` will take care of that for you. One of the nicer features of GDS files is their concept of CELLs. A layout can have several cells, each cell can contain other cells. If the cells are identical, GDS will just use a reference to the cell, saving time and space. In the code above we created a cell `SIMPLE_DEVICE` and added it to our layout. If you are a Cadence EDA (http://www.cadence.com/us/pages/default.aspx) user, you might be a bit confused now. This is because in Cadence

most users just use one big cell for painting. But Cadence EDA (http://www.cadence.com/us/pages/default.aspx) actually supports cells.

Finish the chip

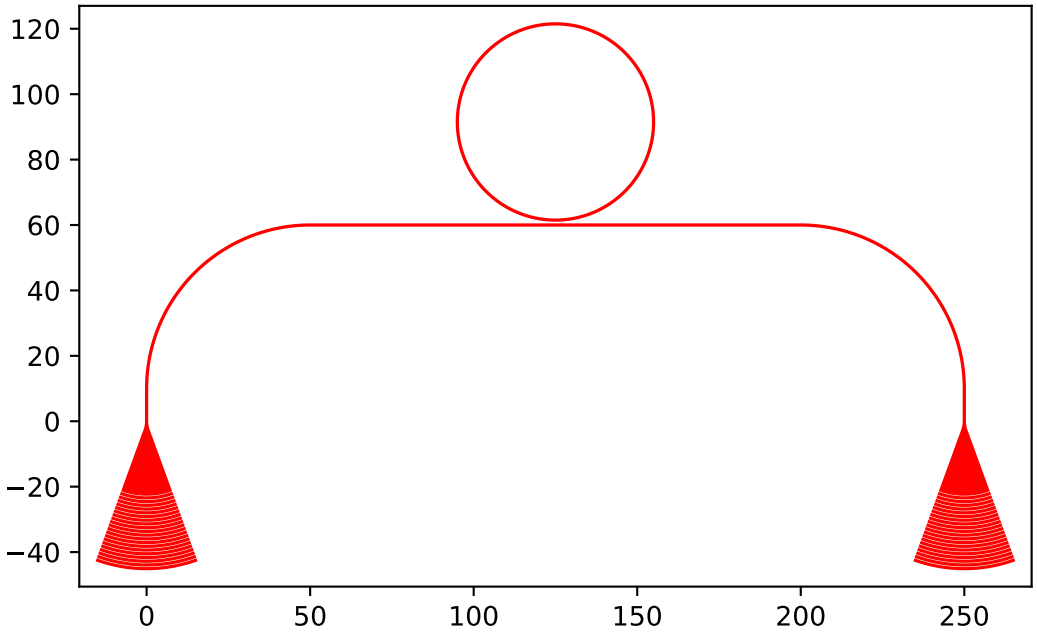
Now, lets run that code by clicking on that green play icon in the top toolbar. You will see a new window showing you what you just designed. Additionally, a new file called `chip.gds` appears in your project folder. The is the GDS file we wanted to create. You can open it in KLayout now.

Exercises

Please also take your time to extend your chip according to the images. You can see one possible solution by clicking on `Source code` above the image.

Insert a resonator

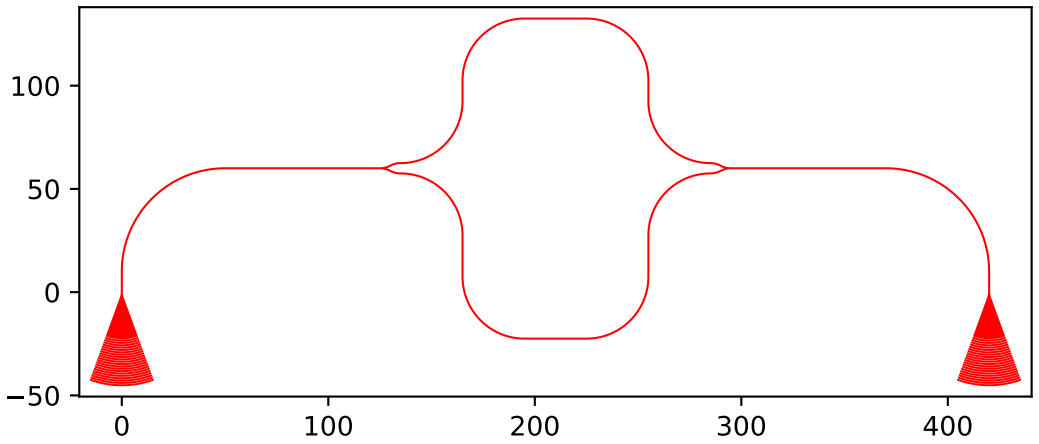
Use *gdshelpers.parts.resonator.RingResonator* to add a ring resonator to your design.



You might also want to play around with the possible extra parameters. Try `race_length`, `res_wg_width`. What happens if the `gap` is *negative*?

Insert a Mach-Zehnder interferometer

You can also easily insert a Mach-Zehnder interferometer, since it is already included in the parts. Try out the *gdshelpers.parts.interferometer.MachZehnderInterferometer* class.



Note, how the interferometer is basically just composed of the parts we used before, except the Y-splitter. This part will be covered in the next device. For now, remember that if you ever plan to create your own part - *MachZehnderInterferometer* is a good place to start looking into the inner workings of `gdshelpers`.

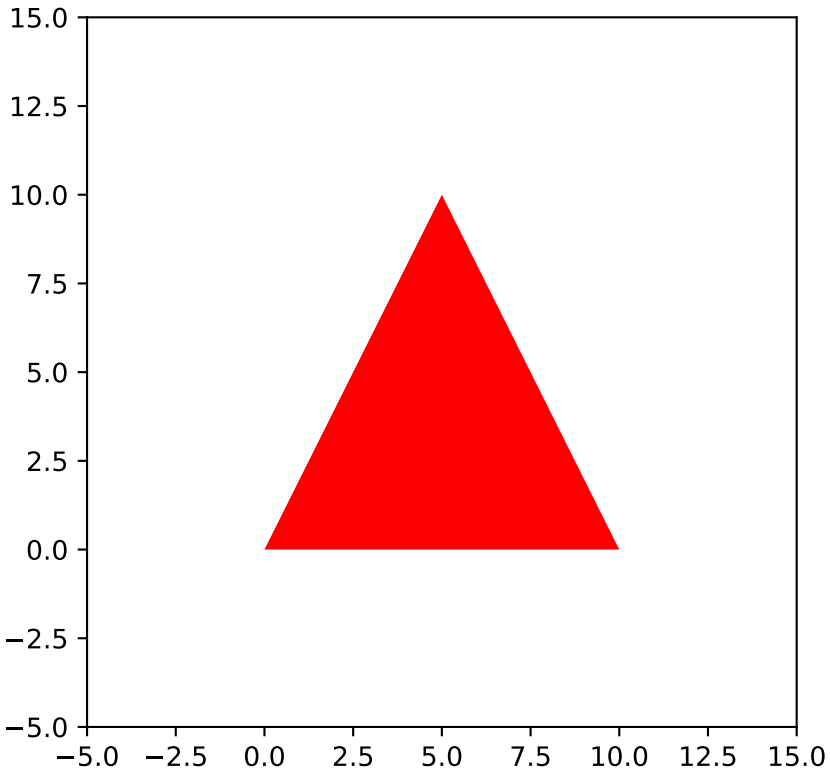
3.2.4 Simple Drawing

While in the beginning it might be enough to use the included parts, you will quickly need to design your own parts and geometries. Remember that you will be using Shapely (http://toblerity.org/shapely/manual.html) to generate your polygons. The only *magic* will be done internally when converting to gdsCAD (http://pythonhosted.org/gdsCAD/).

Simple polygons

Let's start with the most simple polygon one could think of - a triangle! Let the corners be at `(0, 0)`, `(10, 0)` and `(5, 10)`:

```
from gdshelpers.geometry.chip import Cell

from shapely.geometry import Polygon

outer_corners = [(0, 0), (10, 0), (5, 10)]
polygon = Polygon(outer_corners)

cell = Cell('POLYGON')
cell.add_to_layer(1, polygon)
cell.show()
```



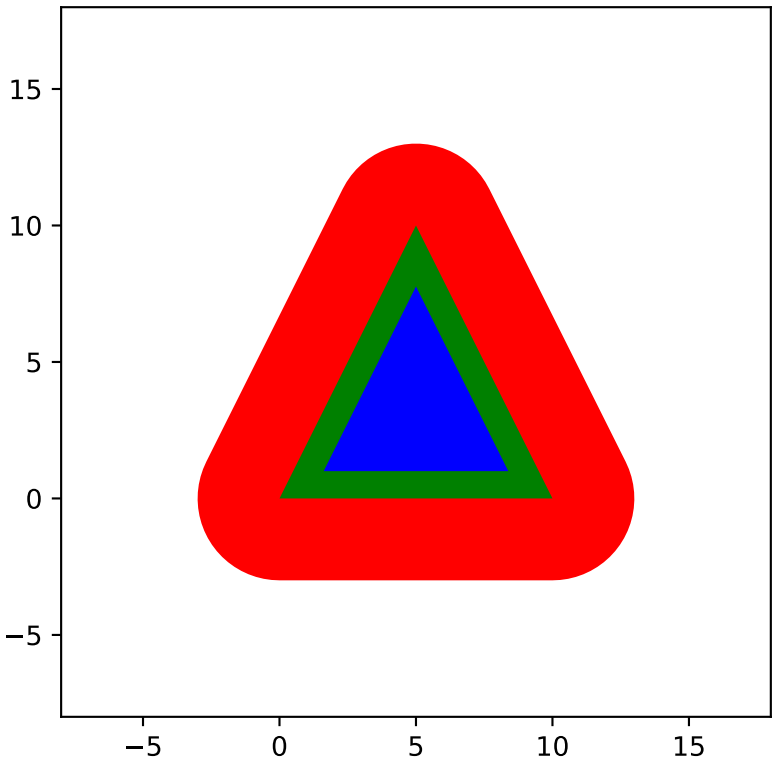
That's simple, right? We `import` the Polygon from `shapely.geometry` just as we did with `pi` in the previous example. A Shapely polygon always has a outer hull and optional holes - which we did not use here.

You can easily build more complex polygons. But make sure, your outer lines do not cross because such polygons are not valid. One simple trick to *clean* such a invalid polygon is the `obj.buffer(0)` command. In this case, a self-intersecting polygon such as the classic "bowtie" will be split into two polygons. More recent versions of gdshelpers will try to produce an acceptable output even if the polygon is invalid. You will however still see an error message and it is strongly advised to fix up your code.

Generating a circle

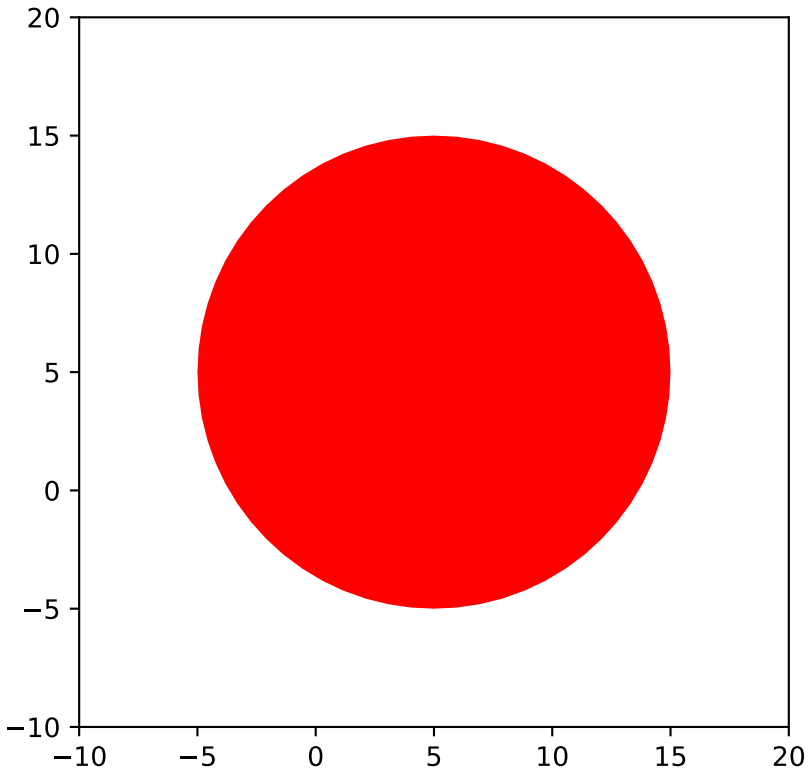
There is a neat trick to generate filled circles: A filled circle is nothing more than a Point, which has been "blown up" in all directions. It turns out that there all Shapely objects have a `buffer()` method. So we could increase the size of our triangle:

```
polygon = Polygon(outer_corners)
polygon_inflated = polygon.buffer(3.)
polygon_deflated = polygon.buffer(-1.)
```



Naturally, this also works for `Points`:

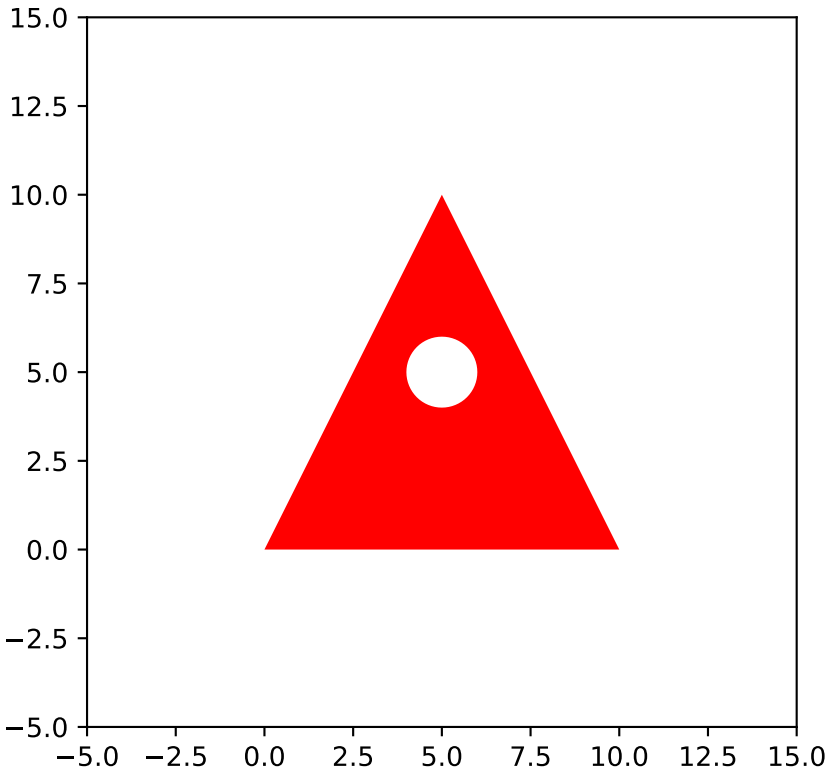
```
point = Point(5, 5)
point_inflated = point.buffer(1.)
```



Boolean operations

Shapely (http://toblerity.org/shapely/manual.html) includes a lot of boolean operations like `a.difference(b)`, `a.intersection(b)`, `a.symmetric_difference(b)` as well as `a.union(b)`. The names should be self-explanatory, right? So let's cut a hole into our triangle:

```
from shapely.geometry import Polygon, Point
from gdshelpers.geometry.chip import Cell

outer_corners = [(0, 0), (10, 0), (5, 10)]
polygon = Polygon(outer_corners)

point = Point(5, 5)
point_inflated = point.buffer(1)

cut_polygon = polygon.difference(point_inflated)

cell = Cell('POLYGON')
cell.add_to_layer(1, cut_polygon)
cell.show()
```



Using parts for polygon operation

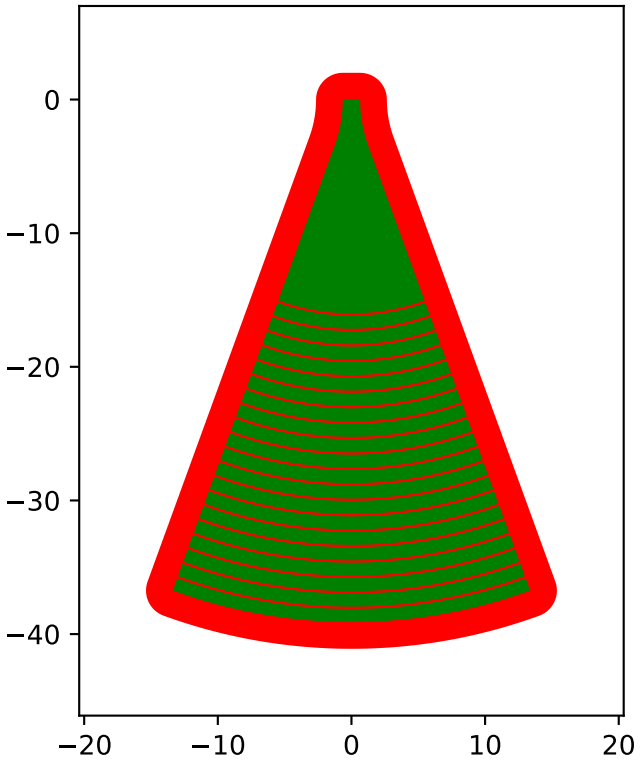
Ok, so for now we used a Shapely object and its methods for polygon manipulation. Naturally, you can also use parts. When you go back to *Software design and intended usage* you will see that all parts provide a `get_shapely_object()` function. So this function will return a Shapely object which you can manipulate further:

```
from math import pi
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler
from gdshelpers.parts.resonator import RingResonator

coupler_params = {
    'width': 1.3,
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

coupler = GratingCoupler.make_traditional_coupler(origin=(0, 0), **coupler_params)
coupler_shapely = coupler.get_shapely_object()

# Do the manipulation
buffered_coupler_shapely = coupler_shapely.buffer(2)
```

(continues on next page)

(continued from previous page)

```
cell = Cell('CELL')
cell.add_to_layer(1, buffered_coupler_shapely)
cell.add_to_layer(2, coupler_shapely)
cell.show()
```



Using multiple parts and/or Shapely objects

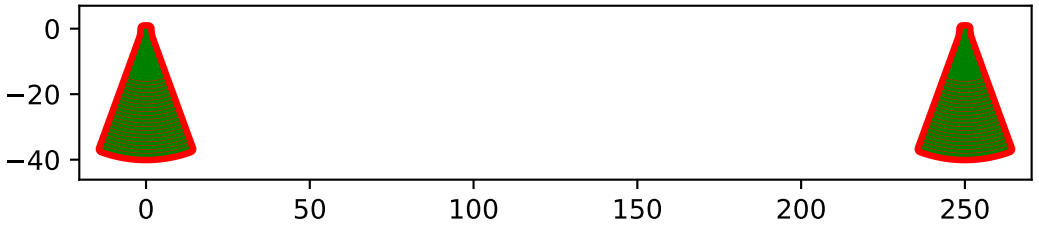
Now, most of the times you will have to deal with *multiple* parts and maybe Shapely objects. Instead of calling `get_shapely_object()` for each part and building the common union of all parts, the *gdshelpers.geometry.geometric_union()* function provides a fast way of merging a *list* (or other kind of iterable) into one big Shapely container:

```
coupler_params = {
    'width': 1.3,
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

coupler1 = GratingCoupler.make_traditional_coupler(origin=(0, 0), **coupler_params)
coupler2 = GratingCoupler.make_traditional_coupler(origin=(250, 0), **coupler_params)


both_coupler_shapely = geometric_union([coupler1, coupler2])
```

(continues on next page)

(continued from previous page)

```
# Do the manipulation
buffered_both_coupler_shapely = coupler_shapely.buffer(2)
```



3.2.5 Sweeping a parameter space

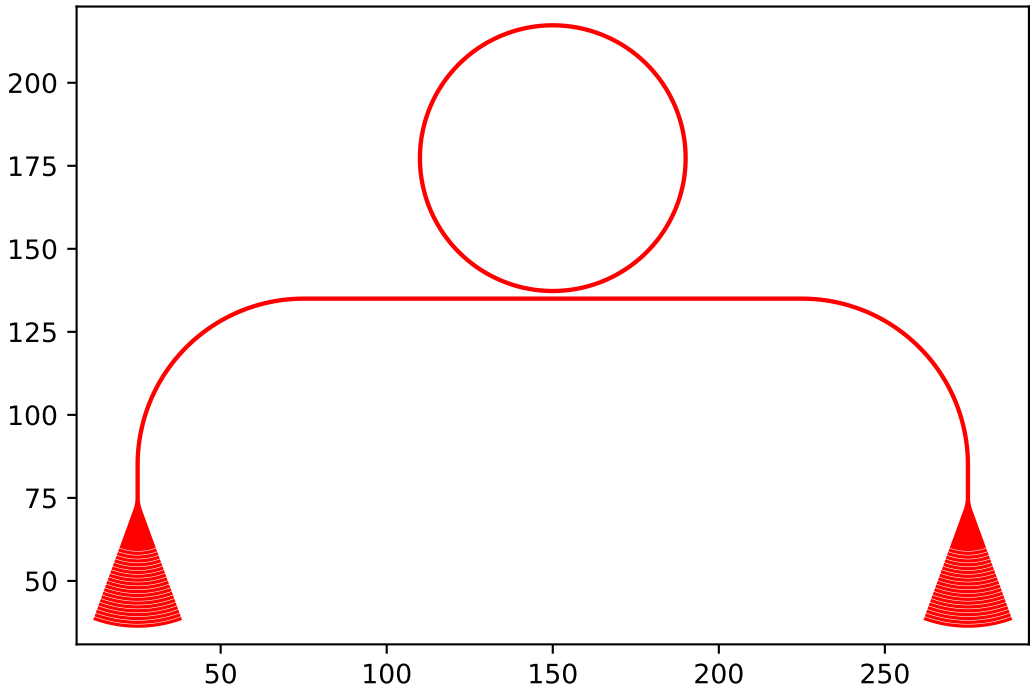
When you start designing your first chips you will probably have a simple chip design like the one introduced in *A simple chip*. Let's say you already got a nice program which generates a cell with your device:

```
import numpy as np
from math import pi
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler
from gdshelpers.parts.resonator import RingResonator

coupler_params = {
    'width': 1.3,
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

def generate_device_cell(resonator_radius, resonator_gap, origin=(25, 75)):
    left_coupler = GratingCoupler.make_traditional_coupler(origin, **coupler_params)
    wg1 = Waveguide.make_at_port(left_coupler.port)
    wg1.add_straight_segment(length=10)
    wg1.add_bend(-pi/2, radius=50)
    wg1.add_straight_segment(length=75)

    ring_res = RingResonator.make_at_port(wg1.current_port, gap=resonator_gap,␣
↪radius=resonator_radius)

    wg2 = Waveguide.make_at_port(ring_res.port)
    wg2.add_straight_segment(length=75)
    wg2.add_bend(-pi/2, radius=50)
    wg2.add_straight_segment(length=10)
    right_coupler = GratingCoupler.make_traditional_coupler_at_port(wg2.current_port,␣
↪**coupler_params)

    cell = Cell('SIMPLE_RES_DEVICE')
    cell.add_to_layer(1, left_coupler, wg1, ring_res, wg2, right_coupler)
    return cell
```

(continues on next page)

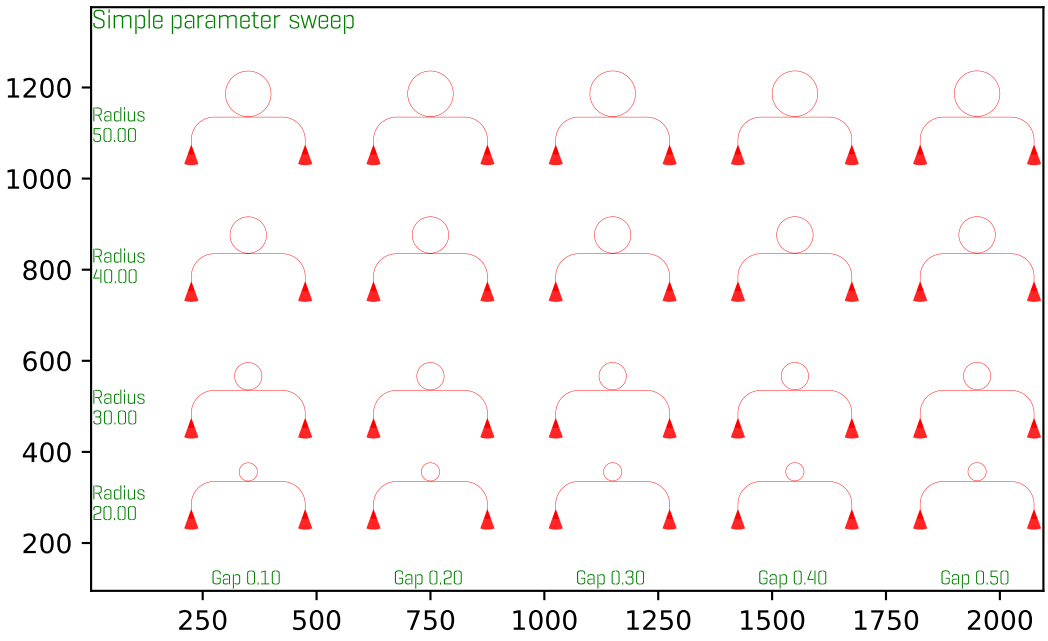
(continued from previous page)

```
example_device = generate_device_cell(40., 1.)
example_device.show()
```



Note, how the `generate_device_cell` creates one single gdsCAD cell per device. For now we just picked two random values for the resonator radius and the gap between the waveguides. Now, how do we sweep over several parameters and add them to those nice layouts with labels and a frame around it? You could create a new cell and add a reference to the device cells to it. While adding a cell reference in gdsCAD you can also specify transformations like translation and/or rotation. For now, we are just after a simple standard layout, so we can use the *GridLayout* included in `gdshelpers`:

```
layout = GridLayout(title='Simple parameter sweep')
radii = np.linspace(10, 20, 4)
gaps = np.linspace(0.1, 0.5, 5)

# Add column labels
layout.add_column_label_row(('Gap %0.2f' % gap for gap in gaps), row_label='')

for radius in radii:
    layout.begin_new_row('Radius\n%0.2f' % radius)
    for gap in gaps:
        layout.add_to_row(generate_device_cell(radius, gap))

layout_cell, mapping = layout.generate_layout()
layout_cell.show()
```



By default *GridLayout* will place all devices on a regular grid as close as possible - while maintaining a minimum spacing and aligning to write fields. If your original cell was optimized to write fields (this one was not), your generated layout will also be within the write fields. To profit from this, assume your write field starts at `(0, 0)`. This is valid, even if your electron beam write starts its write field at the top left structure. The frame of the layout will force a correct write field in this case. If you worked with older versions of gdshelper, you might have used `TiledLayout` which was the initial attempt on a device layout manager. Unfortunately, it proved to be unflexible. If you want to pack your devices as close as possible in the x-direction. Pass `tight=True` to the GridLayout constructor. Region layers can either be placed per cell, or per layout. The region layer behaviour can be changed with the `region_layer_type` and `region_layer_on_labels` parameters. Refere to the `TiledLayout` documentation for more details. Also note, that `GridLayout.generate_layout()` returns *two* values. We have only used the first value `layout_cell`. The value in `mapping` will tell you where each device was placed. To make use of this, you have to pass a unique id when calling `add_to_row`.

3.2.6 Generating electron beam lithography markers

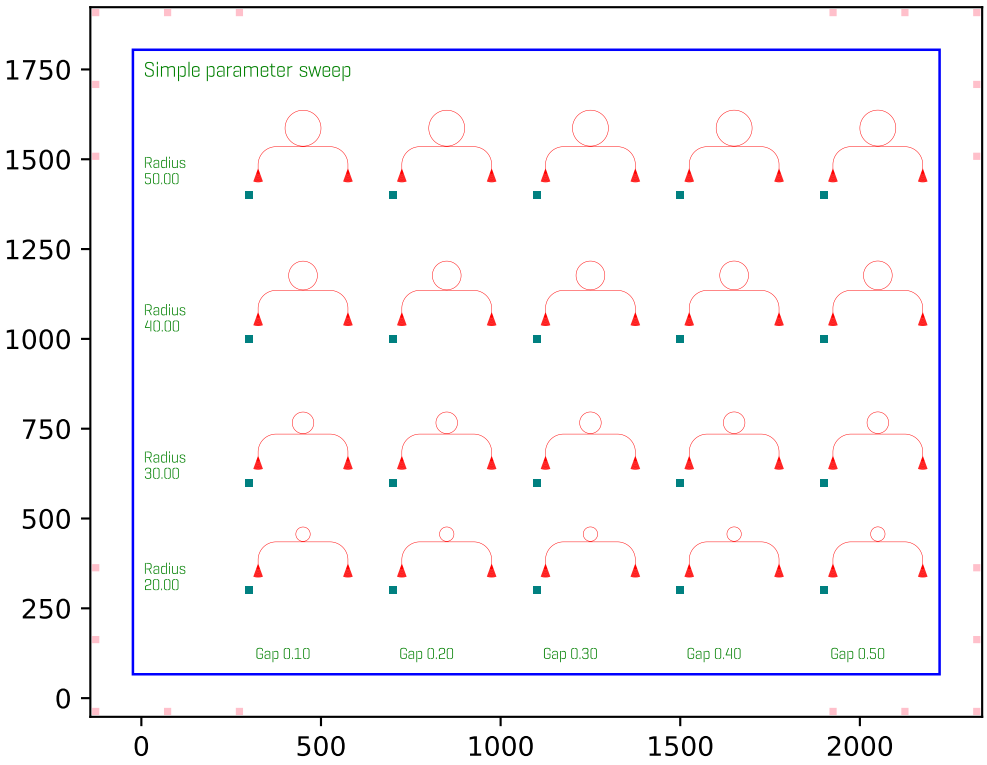
When writing several layers with electron beam lithography, markers are needed to align these layers. There is a class in gdshelpers that will help you to generate these markers. Note that at the moment only square markers can be found in the library. However, other types of markers can easily be added by writing an own `frame_generator` and then using the same method. Here is one example how global and local markers can be added:

```
import numpy as np

from math import pi
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler
from gdshelpers.parts.resonator import RingResonator
from gdshelpers.layout import GridLayout
from gdshelpers.parts.marker import SquareMarker
from gdshelpers.geometry.ebl_frame_generators import raith_marker_frame
```

(continues on next page)

(continued from previous page)

```
coupler_params = {
    'width': 1.3,
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

def generate_device_cell(resonator_radius, resonator_gap, origin=(25, 75)):
    left_coupler = GratingCoupler.make_traditional_coupler(origin, **coupler_params)
    wg1 = Waveguide.make_at_port(left_coupler.port)
    wg1.add_straight_segment(length=10)
    wg1.add_bend(-pi / 2, radius=50)
    wg1.add_straight_segment(length=75)

    ring_res = RingResonator.make_at_port(wg1.current_port, gap=resonator_gap,␣
↪radius=resonator_radius)

    wg2 = Waveguide.make_at_port(ring_res.port)
    wg2.add_straight_segment(length=75)
    wg2.add_bend(-pi / 2, radius=50)
    wg2.add_straight_segment(length=10)
    right_coupler = GratingCoupler.make_traditional_coupler_at_port(wg2.current_port,␣
↪**coupler_params)

    cell = Cell('SIMPLE_RES_DEVICE r={:.1f} g={:.1f}'.format(resonator_radius,␣
↪resonator_gap))
    cell.add_to_layer(1, left_coupler, wg1, ring_res, wg2, right_coupler)
    cell.add_ebl_marker(layer=9, marker=SquareMarker(origin=(0, 0), size=20))
    return cell


layout = GridLayout(title='Simple parameter sweep', frame_layer=0, text_layer=2,␣
↪region_layer_type=None)
radii = np.linspace(20, 50, 4)
gaps = np.linspace(0.1, 0.5, 5)

# Add column labels
layout.add_column_label_row(('Gap %0.2f' % gap for gap in gaps), row_label='')

for radius in radii:
    layout.begin_new_row('Radius\n%0.2f' % radius)
    for gap in gaps:
        layout.add_to_row(generate_device_cell(radius, gap))

layout_cell, mapping = layout.generate_layout()
layout_cell.add_frame(frame_layer=8, line_width=7)
layout_cell.add_ebl_frame(layer=10, frame_generator=raith_marker_frame, n=2)
layout_cell.show()
```



First of all, we can add local EBL markers with `add_ebl_marker` and a defined position. Secondly, global markers are added with `add_ebl_frame`, and the number of markers per corner can be adjusted by changing the parameter `n`. In addition to the EBL markers, we added a frame around our structures with `add_frame`.

3.2.7 Slot waveguides and mode converters

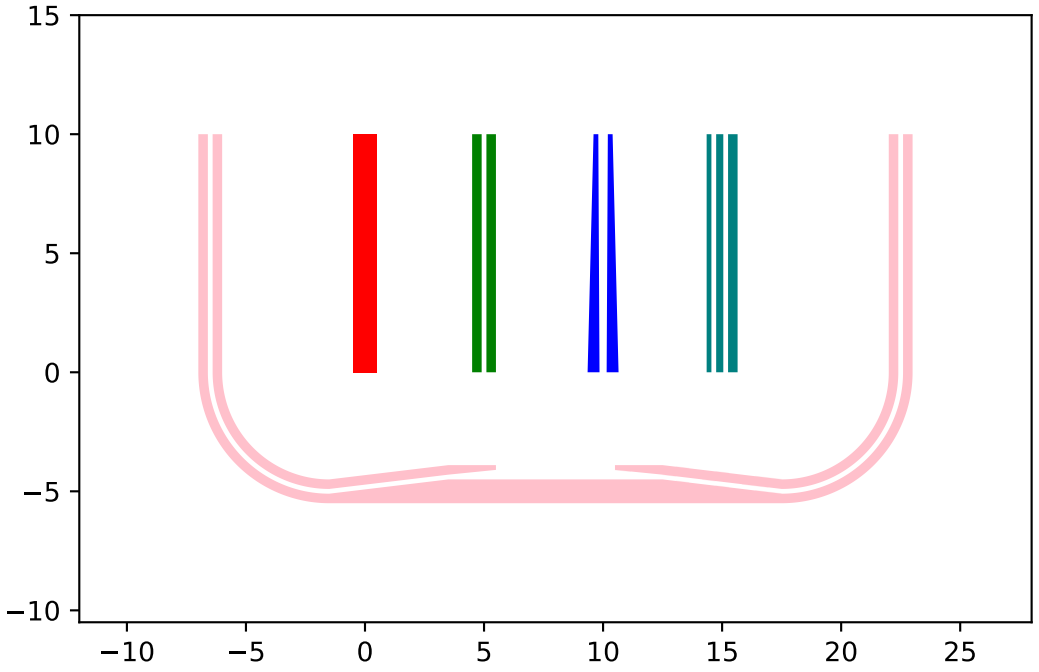
So far, only strip waveguides have been used. However, gdshelpers includes also slot waveguides and strip-to-slot mode converters. Some examples are shown below:

```
import numpy as np

from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.mode_converter import StripToSlotModeConverter
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.port import Port

# waveguide 1: strip waveguide
wg_1 = Waveguide.make_at_port(Port(origin=(0, 0), angle=np.pi / 2, width=1))
wg_1.add_straight_segment(length=10)

# waveguide 2: slot waveguide
wg_2 = Waveguide.make_at_port(Port(origin=(5, 0), angle=np.pi / 2, width=[0.4, 0.2, 0.
↪4]))
wg_2.add_straight_segment(length=10)

# waveguide 3: slot waveguide with tapering
wg_3 = Waveguide.make_at_port(Port(origin=(10, 0), angle=np.pi / 2, width=[0.5, 0.3,␣
↪0.5]))
```

(continues on next page)

(continued from previous page)

```
wg_3.add_straight_segment(length=10, final_width=[0.2, 0.4, 0.2])

# waveguide 4: slot waveguide with three rails and two slots
wg_4 = Waveguide.make_at_port(Port(origin=(15, 0), angle=np.pi / 2, width=[0.2, 0.2,␣
↪0.3, 0.2, 0.4]))
wg_4.add_straight_segment(length=10)

# waveguide 5: slot waveguide with bends and strip to slot mode converter
wg_5_1 = Waveguide.make_at_port(Port(origin=(-6.5, 10), angle=-np.pi / 2, width=[0.4,␣
↪0.2, 0.4]))
wg_5_1.add_straight_segment(length=10)
wg_5_1.add_bend(angle=np.pi / 2, radius=5)
mc_1 = StripToSlotModeConverter.make_at_port(port=wg_5_1.current_port, taper_length=5,
↪ final_width=1,
                                             pre_taper_length=2, pre_taper_width=0.2)
wg_5_2 = Waveguide.make_at_port(port=mc_1.out_port)
wg_5_2.add_straight_segment(5)
mc_2 = StripToSlotModeConverter.make_at_port(port=wg_5_2.current_port, taper_length=5,
↪ final_width=[0.4, 0.2, 0.4],
                                             pre_taper_length=2, pre_taper_width=0.2)
wg_5_3 = Waveguide.make_at_port(port=mc_2.out_port)
wg_5_3.add_bend(angle=np.pi / 2, radius=5)
wg_5_3.add_straight_segment(length=10)

cell = Cell('Cell')
cell.add_to_layer(1, wg_1)  # red
cell.add_to_layer(2, wg_2)  # green
cell.add_to_layer(3, wg_3)  # blue
cell.add_to_layer(4, wg_4)  # jungle green
cell.add_to_layer(5, wg_5_1, mc_1, wg_5_2, mc_2, wg_5_3) # pink
cell.show()
```



The routing is very similar to the routing of a strip waveguide, meaning that a port (origin, angle and width) has to be defined, and waveguides elements can be added from this port. The only difference is that the width of the waveguide is not given by a scalar, as shown in the case of waveguide 1, but by an array, usually with an odd number of elements. In this array, each element with an odd number denotes the width of a rail (waveguide 2), while each element with an even number denotes the width of the slot between two rails. As in the case of strip waveguides, one can make use of tapering (waveguide 3), bends (waveguide 5_1 and 5_3) and all other kinds of routing functions that are available in the *Waveguide* class.

Using the *StripToSlotModeConverter* class, strip to slot mode converters can added, which allow for a transition from a strip waveguide to a slot waveguide and vice versa. To create this element, five parameters have to be defined: The current port (origin, angle and width), the length of the taper, the final width and the width and length of the pre taper. If the current port width is a scalar and the final width is an array with three elements (two rails and one slot), a strip to slot mode converter is created. In the opposite case, a slot to strip mode converter is defined.

3.2.8 More advanced waveguide features

In the previous chapter, the waveguide part was already introduced and commonly used. While you might already be satisfied with what you got there, there are still a lot more useful hidden features.

Chaining of `add_` calls

You will find yourself often calling several successive `add_` type methods which will use lots of source code space. Code such as this:

```
wg = Waveguide.make_at_port(left_coupler.port)
wg.add_straight_segment(length=10)
wg.add_bend(-pi/2, radius=50)
wg.add_straight_segment(length=150)
wg.add_bend(-pi/2, radius=50)
wg.add_straight_segment(length=10)
```

Can be rewritten by chaining the construction calls:

```
wg = Waveguide.make_at_port(left_coupler.port)
wg.add_straight_segment(length=10).add_bend(-pi/2, radius=50)
wg.add_straight_segment(length=150).add_bend(-pi/2, radius=50)
wg.add_straight_segment(length=10)
```

This works, since all `add_` type methods return the modified waveguide object itself again, which you can then call just as you do with `wg.`.

Length measurements

Sometimes it is important to get the length of a Waveguide. Simply query `.length` to get the length of a waveguide. This even works for parameterized paths, but naturally it will only be a numerical approximation.

Automatic routing

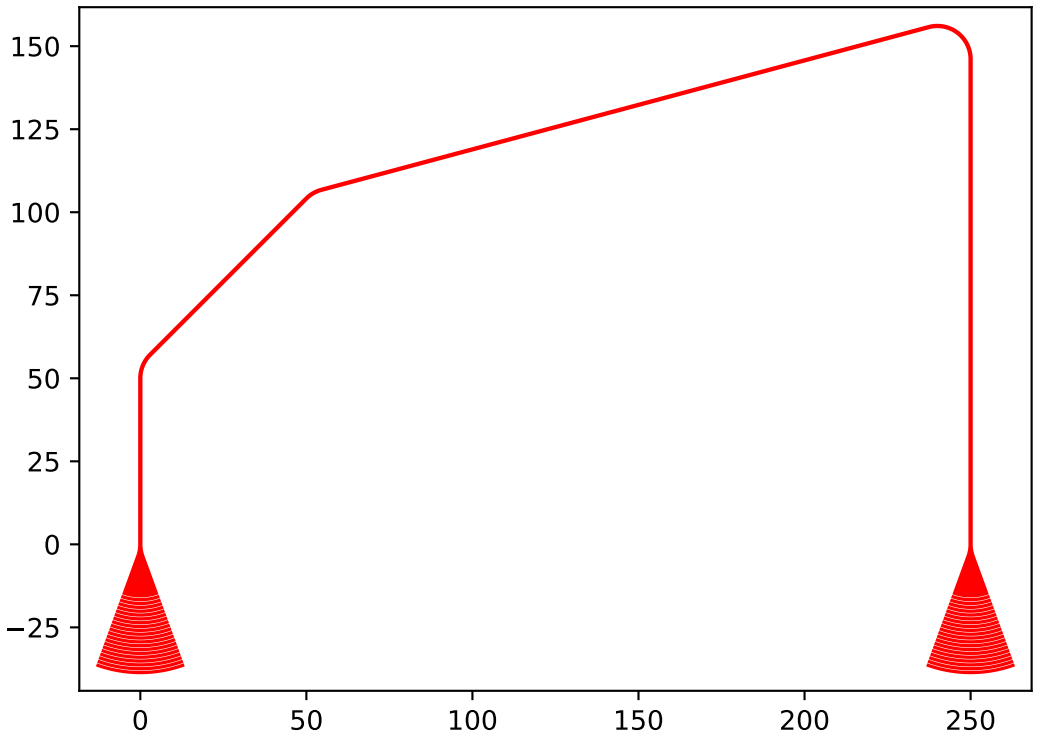
Lot's of times you will want to connect two points, but you always have to calculate the distance and factor in the bending radius etc. Since this is boring work and prone to error, a lot of useful routing functions are include in the *Waveguide* class. Available functions are:

- `Waveguide.add_bezier_to()` and `Waveguide.add_bezier_to_port()`

- `Waveguide.add_route_single_circle_to()` and `Waveguide.add_route_single_circle_to_port()`
- `Waveguide.add_straight_segment_to_intersection()`
- `Waveguide.add_straight_segment_until_level_of_port()`
- `Waveguide.add_straight_segment_until_x()` and `Waveguide.add_straight_segment_until_y()`

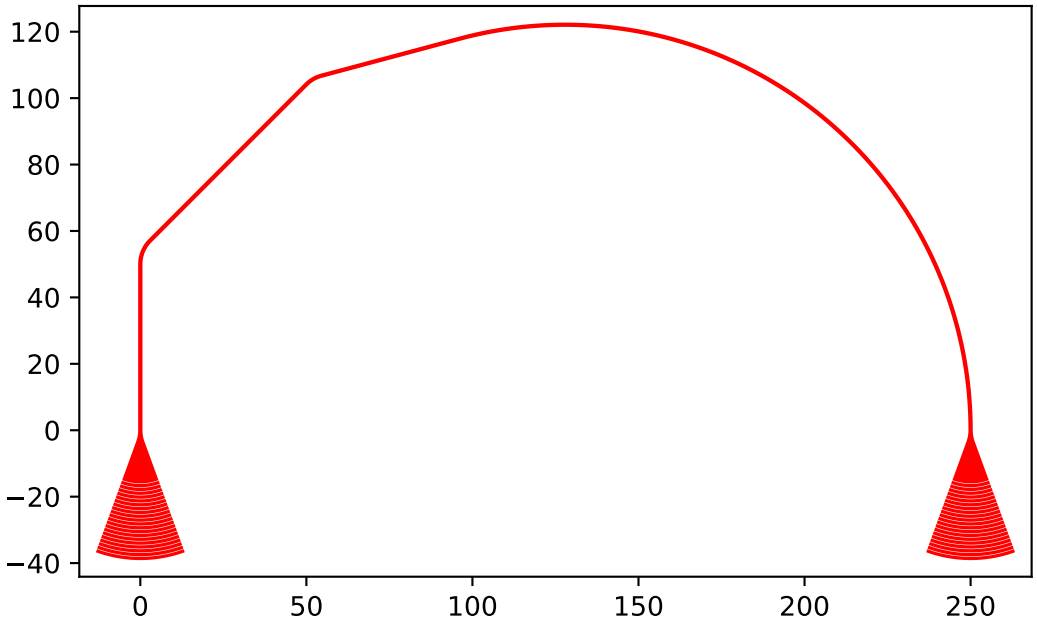
It's probably best explained by an example. But if your are interested you can also check out the *Waveguide* class documentation:

```
import numpy as np
from math import pi
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler

coupler_params = {
    'width': 1.3,
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

left_coupler = GratingCoupler.make_traditional_coupler((0,0), **coupler_params)
right_coupler = GratingCoupler.make_traditional_coupler((250,0), **coupler_params)

wg = Waveguide.make_at_port(left_coupler.port)
wg.add_straight_segment_until_y(50)
wg.add_bend(np.deg2rad(-45), 10)
wg.add_straight_segment_until_x(50)
wg.add_bend(np.deg2rad(-30), 10)
wg.add_route_single_circle_to_port(right_coupler.port, 10)

cell = Cell('SIMPLE_DEVICE')
cell.add_to_layer(1, left_coupler, wg, right_coupler)
cell.show()
```



One other useful feature of `Waveguide.add_route_single_circle_to()` is that it will attempt to use the biggest possible bend radius if no maximal bend radius is specified:

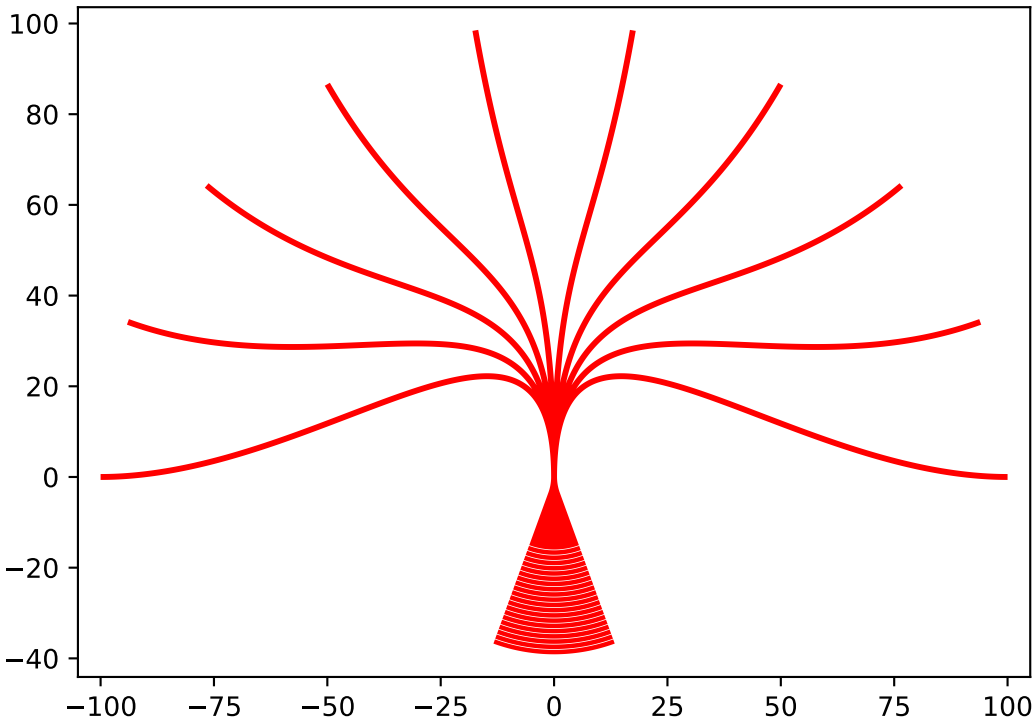
```
wg.add_route_single_circle_to_port(right_coupler.port)
```



If the maximum bend radius is set to zero, you will get a sharp edge.

What we have omitted until now, is Bézier curve routing. This routing is special in the sense that it will give you smooth lines only. There will basically be no straight lines or circles. An example:

```
import numpy as np
from math import pi
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler


coupler_params = {
    'width': 1.3,
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

coupler = GratingCoupler.make_traditional_coupler((0,0), **coupler_params)
wgs = list()
for angle in np.linspace(-np.pi/2, np.pi/2, 10):
    # Calculate the target port
    # We do this by changing the angle of the coupler port and calculating a
    # longitudinal offset. Since the port then points outwards, we invert its␣
↪direction.
    target_port = coupler.port.rotated(angle).longitudinal_offset(100).inverted_
↪direction

    wg = Waveguide.make_at_port(coupler.port)
    wg.add_bezier_to_port(target_port, bend_strength=50)
    wgs.append(wg)

cell = Cell('SIMPLE_DEVICE')
cell.add_to_layer(1, coupler)
cell.add_to_layer(1, *wgs)
cell.show()
```

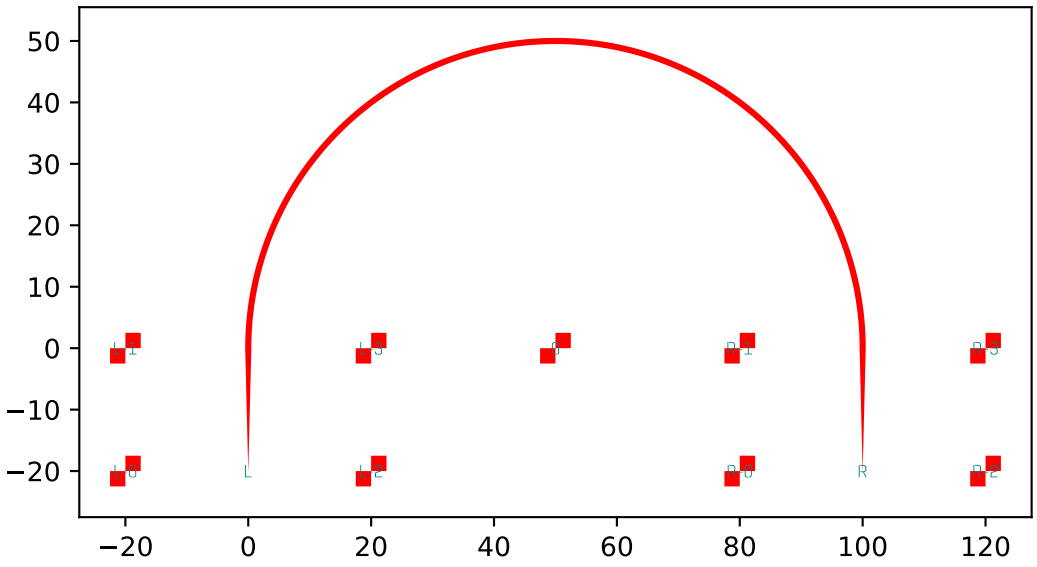


Notice the `bend_strength` parameter of `Waveguide.add_bezier_to_port()`. The heigher the parameter, the smoother the connecting lines will be. But take care: For big values the Bézier curve might intersect with itself which will give you an error. In short, Bézier curves can be very useful to connect to non-trivial points - but they might give you errors on self intersection and are generally quite slow to calculate.

3.2.9 Interfacing 3D-hybrid structures

For interfacing integrated planar circuitry with 3D-hybrid structures, tapers need to be included into the design. In the vicinity of each taper alignment markers need to be included as well, allowing determination of the taper positions using computer vision.

This can simply be done by using the method `Cell.add_dlw_taper_at_port()`. The first parameter defines the name of the taper within the cell. In order to assure unique names, the complete name of the Cell includes the names of the surrounding cells separated by dots (but not the topmost cell, as there's anyway just one). E.g. in the following example, the name of the tapers are defined as *A0.L* and *A0.R*. For each taper four alignment markers are generated automatically around the taper. Each marker name is composed by the name of the taper and an postfix *-X*, where X is a number from 0-3. The exact naming is shown in the layout on the comments layer.

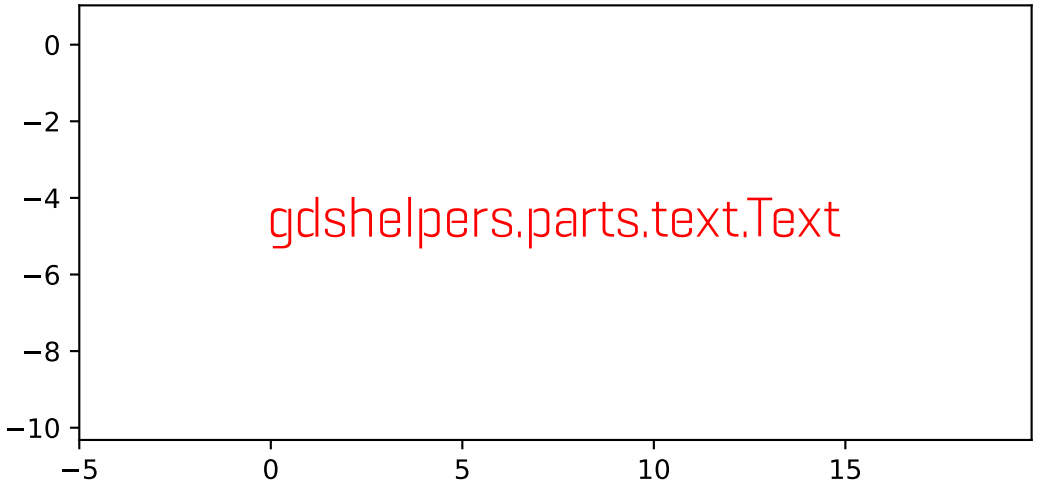


Besides automatically generated markers, the user can also directly add markers to the layout using `add_dlw_marker()` as shown in the example. This is on the one side handy for adding reference markers on the topmost level of the design, allowing for simple names (The marker in the example is just called "0", as it's on the topmost level, there are no cell names as prefixes). On the other hand, manual adding of the tapers is required, if the standard locations of the markers are already used by other elements in the design. By passing *with_tapers=False* as an parameter to `Cell.add_dlw_taper_at_port()`, automatic generation of the markers can be suppressed and the user is required to place the markers.

3.2.10 Fonts

It is always a good idea to label your designs extensively. Naturally, text is also supported in gdshelpers.

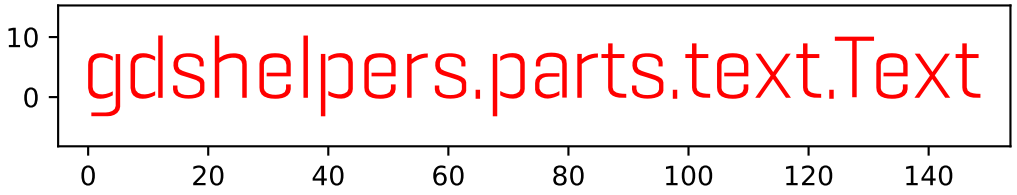
Gdshelpers supports its own font, using pure Shapely objects.



Writing text

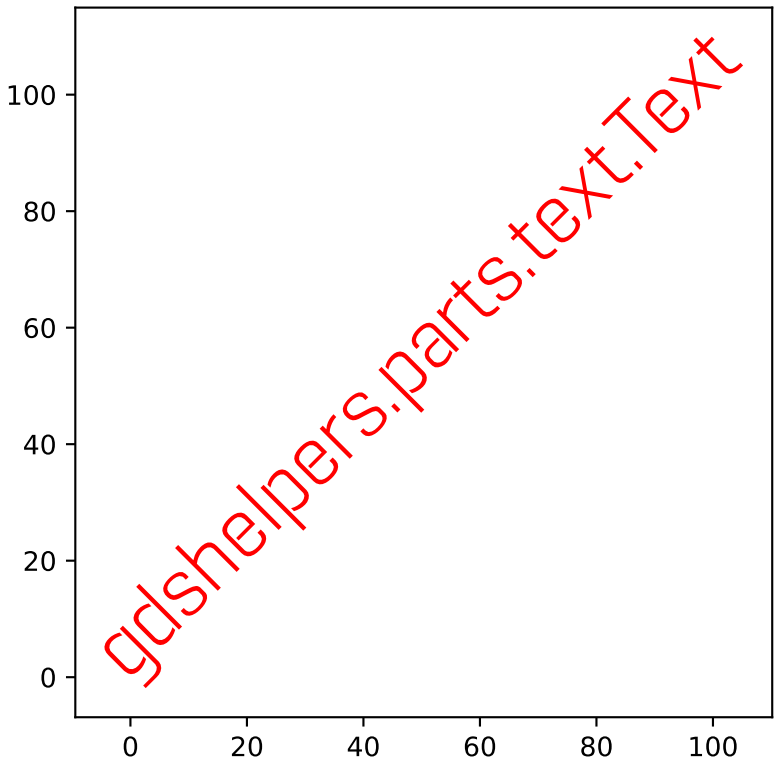
You have heard a lot about different text and label types now. Let's get our hands dirty. The *gdshelpers.parts.text.Text* class behaves like any other part you already now. Typically you pass at least three options: `origin`, the text `height` and the actual `text`:

```
from gdshelpers.parts.text import Text
text = Text([0, 0], 10, 'gdshelpers.parts.text.Text')
```



You can additionally specify an angle:

```
text = Text([0, 0], 10, 'gdshelpers.parts.text.Text', angle=np.pi/4)
```

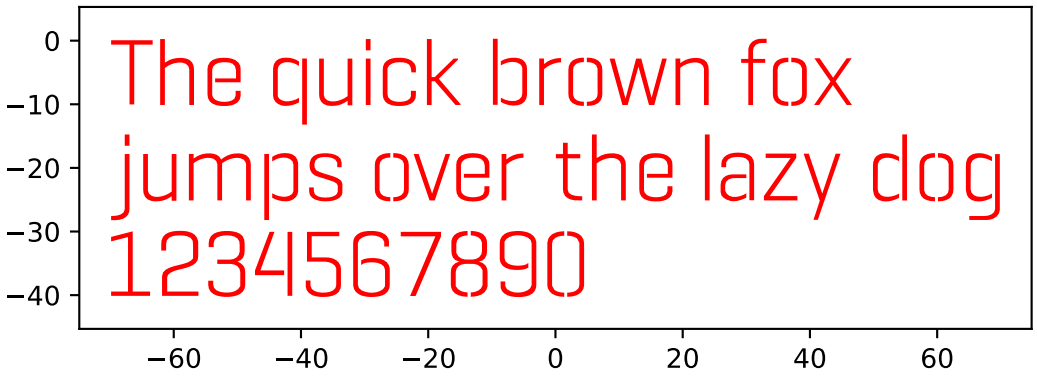


Another handy option is the `alignment` option. It lets you specify the alignment of the text. Alignment can be set independently for the x- and y-axis. Valid options are `left`, `center`, `right` for the x axis and `bottom`, `center`, `top` for the y-axis. So `right-top` will center the text to the upper right corner:

```
text = Text([0, 0], 10, 'top-right', alignment='right-top')
```

Note: You can also write multiple lines at the same time! Simply use the `\n` character:

```
text = Text([0, 0], 10, 'The quick brown fox\njumps over the lazy dog\n1234567890',
            alignment='center-top')
```



3.2.11 Final words

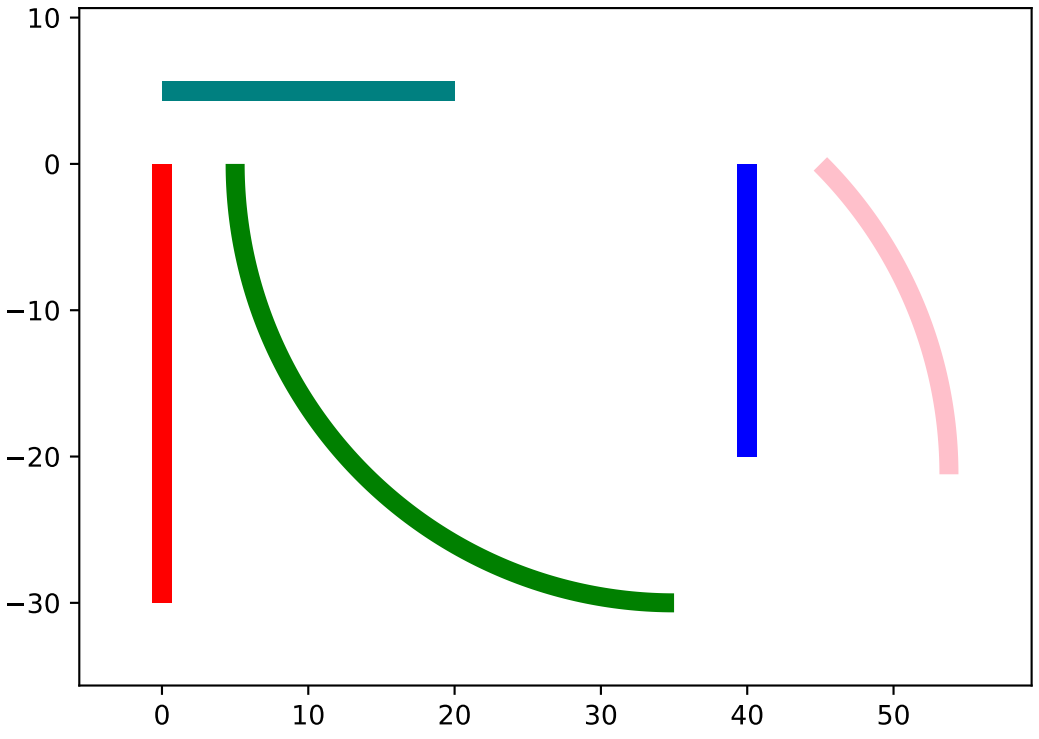
We now reached the end of this tutorial. In the next chapters we'll focus on the growling list of parts implemented in this library.

3.3 Parts

3.3.1 Waveguides

Waveguides are the most basic structures of integrated photonics circuits. To make the routing of these structures as easy as possible, gdshelpers has many different ways of creating waveguides. First of all, we have to define a port, meaning we have to give the origin, width and rotation of our waveguide. Now we can start with our routing. All in all the *Waveguide* class, which can be found in the *waveguide* package, contains 13 ways how to define waveguides. Let us start with the five simplest functions:

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide

wg_1 = Waveguide.make_at_port(Port((0, 0), angle=-np.pi / 2, width=1.3))
wg_1.add_straight_segment(length=30)

wg_2 = Waveguide.make_at_port(Port((5, 0), angle=-np.pi / 2, width=1.3))
wg_2.add_bend(angle=np.pi / 2, radius=30)

wg_3 = Waveguide.make_at_port(Port((40, 0), angle=-np.pi / 2, width=1.3))
wg_3.add_straight_segment_until_y(y=-20)

wg_4 = Waveguide.make_at_port(Port((0, 5), angle=0, width=1.3))
wg_4.add_straight_segment_until_x(x=20)

wg_5 = Waveguide.make_at_port(Port((45, 0), angle=-np.pi / 4, width=1.3))
wg_5.add_arc(final_angle=-np.pi / 2, radius=30)

cell = Cell('CELL')
cell.add_to_layer(1, wg_1)  # red
cell.add_to_layer(2, wg_2)  # green
cell.add_to_layer(3, wg_3)  # blue
cell.add_to_layer(4, wg_4)  # teal
cell.add_to_layer(5, wg_5)  # pink
cell.show()
```

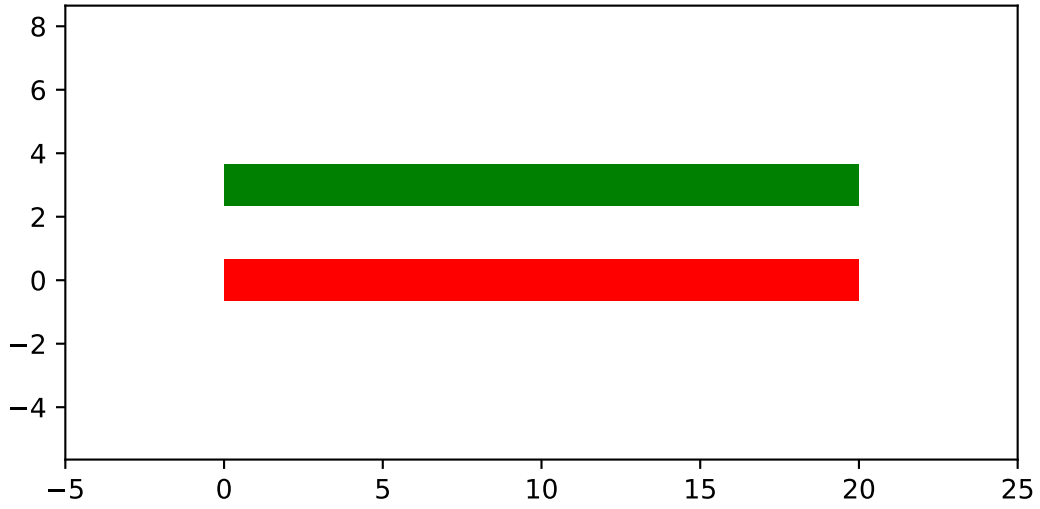


The most basic function is *add_straight_segment*, which adds a waveguide with a given length. Apart from the length we can also define the final width of the waveguide, which is not done in this example. To add bends to our routing,

we can make use of the function *add_bend*. Parameters are the bend angle (as always in radiant) and the bend radius. Sometimes we don't know how long a waveguide has to be or the starting position is unknown and we want to end at a certain position. In this case we can use the *add_straight_segment_until_x/y* function, which adds a waveguide with a length such that it terminates at the defined x/y position. Similarly, if the starting angle is not known and we want to add a bend which terminates in a certain angle, we can use the *add_arc* function.

Note that there are two ways how to start a waveguide. First, we can call the *make_at_port* function, as it was done before. In addition, we can also initialize the waveguide by calling the constructor of the *waveguide* class:

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts import Port

waveguide_1 = Waveguide.make_at_port(Port(origin=(0, 0), angle=0, width=1.3))
waveguide_1.add_straight_segment(length=20)

waveguide_2 = Waveguide(origin=(0, 3), angle=0, width=1.3)
waveguide_2.add_straight_segment(length=20)

cell = Cell('CELL')
cell.add_to_layer(1, waveguide_1)   # red
cell.add_to_layer(2, waveguide_2)   # green
cell.show()
```



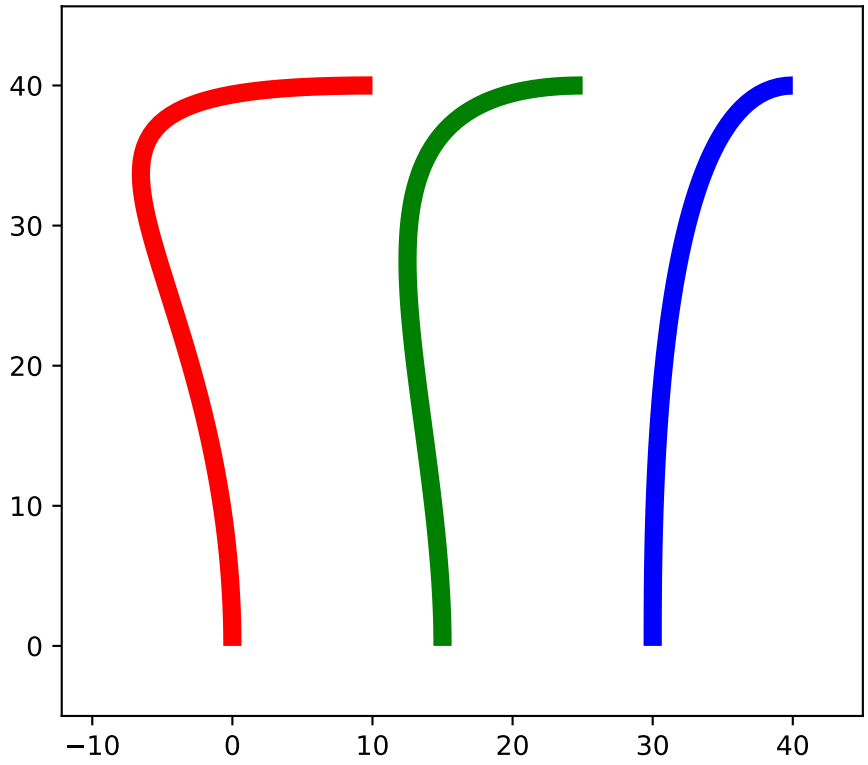
The library contains also more complex routing like Bezier curves oder parameterized paths. One way to create such a path is using the *add_bezier_to function*. In this case we have to define the final coordinates, the final angle and the bend strength of the curve.

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide

wg_1 = Waveguide.make_at_port(Port((0, 0), angle=np.pi / 2, width=1.3))
wg_1.add_bezier_to(final_coordinates=(10, 40), final_angle=0, bend_strength=30)

wg_2 = Waveguide.make_at_port(Port((15, 0), angle=np.pi / 2, width=1.3))
```

(continues on next page)

(continued from previous page)

```
wg_2.add_bezier_to(final_coordinates=(25, 40), final_angle=0, bend_strength=20)

wg_3 = Waveguide.make_at_port(Port((30, 0), angle=np.pi / 2, width=1.3))
wg_3.add_bezier_to(final_coordinates=(40, 40), final_angle=0, bend_strength=10)

cell = Cell('CELL')
cell.add_to_layer(1, wg_1)  # red
cell.add_to_layer(2, wg_2)  # green
cell.add_to_layer(3, wg_3)  # blue
cell.show()
```



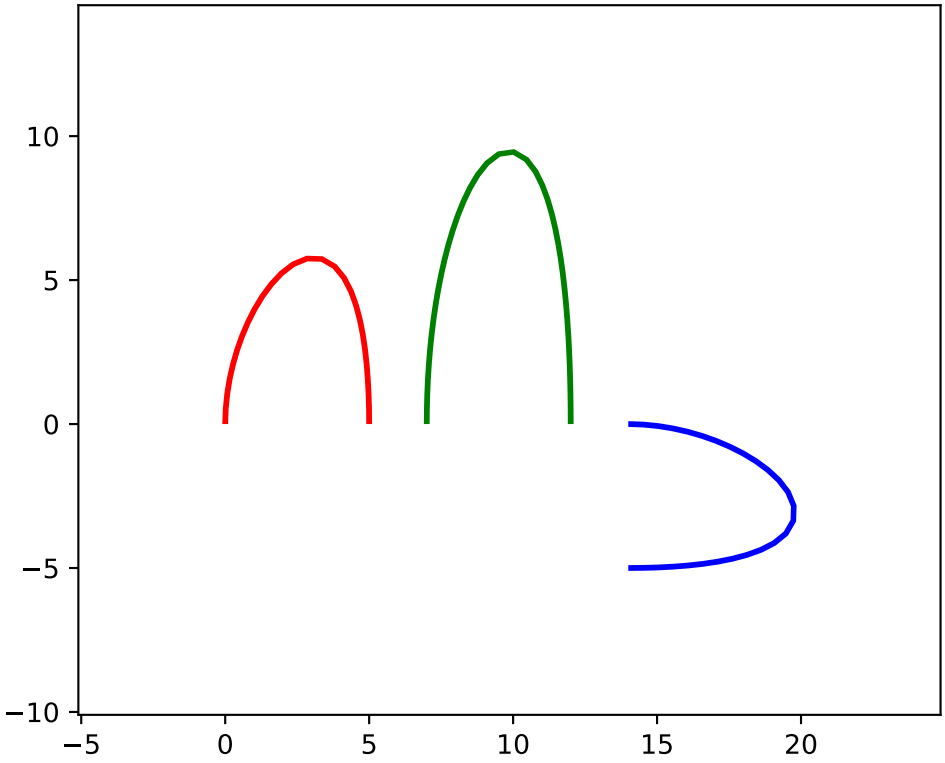
A second way is using the *add_cubic_bezier_path*, in which case the curve can be shaped with four control points.

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide

wg_1 = Waveguide.make_at_port(Port(origin=(0, 0), angle=0, width=0.2))
wg_1.add_cubic_bezier_path(p0=(0, 0), p1=(0, 5), p2=(5, 10), p3=(5, 0))

wg_2 = Waveguide.make_at_port(Port(origin=(7, 0), angle=0, width=0.2))
wg_2.add_cubic_bezier_path(p0=(0, 0), p1=(0, 10), p2=(5, 15), p3=(5, 0))

wg_3 = Waveguide.make_at_port(Port(origin=(14, 0), angle=-np.pi / 2, width=0.2))
wg_3.add_cubic_bezier_path(p0=(0, 0), p1=(0, 5), p2=(5, 10), p3=(5, 0))
```

(continues on next page)

(continued from previous page)

```
cell = Cell('CELL')
cell.add_to_layer(1, wg_1)  # red
cell.add_to_layer(2, wg_2)  # green
cell.add_to_layer(3, wg_3)  # blue
cell.show()
```



At this point we are not going into much more detail about the mathematical background and the meaning of the four points which define the Bezier curve, since enough good literature can be found the in the internet. If you are interested, read it ;)

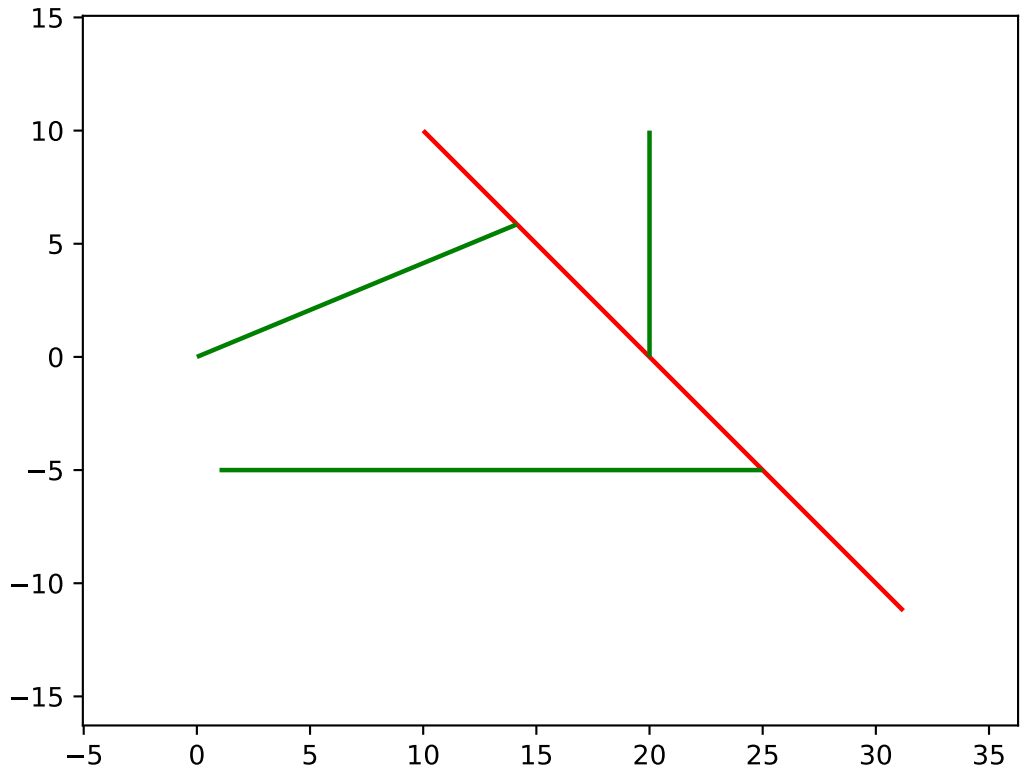
The last function to create waveguides is *add_straight_segment_to_intersection*. From a port a straight line is created, which continues as long as it does not intersect with a second line. This intersection line is defined in the function *add_straight_segment_to_intersection*. Note that this line continues in both directions indefinitely long. For visualization purposes, this intersection line is added in red to our design.

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide

int_line = Waveguide.make_at_port(Port(origin=(10, 10), angle=-np.pi / 4, width=0.2))
int_line.add_straight_segment(length=30)

wg_1 = Waveguide.make_at_port(Port(origin=(0, 0), angle=np.pi / 8, width=0.2))
wg_1.add_straight_segment_to_intersection(line_origin=(10, 10), line_angle=-np.pi / 4)

wg_2 = Waveguide.make_at_port(Port(origin=(1, -5), angle=0, width=0.2))
```

(continues on next page)

(continued from previous page)

```
wg_2.add_straight_segment_to_intersection(line_origin=(10, 10), line_angle=-np.pi / 4)

wg_3 = Waveguide.make_at_port(Port(origin=(20, 10), angle=-np.pi / 2, width=0.2))
wg_3.add_straight_segment_to_intersection(line_origin=(10, 10), line_angle=-np.pi / 4)

cell = Cell('CELL')
cell.add_to_layer(1, int_line)  # red
cell.add_to_layer(2, wg_1)  # green
cell.add_to_layer(2, wg_2)  # green
cell.add_to_layer(2, wg_3)  # green
cell.show()
```



Slot waveguides

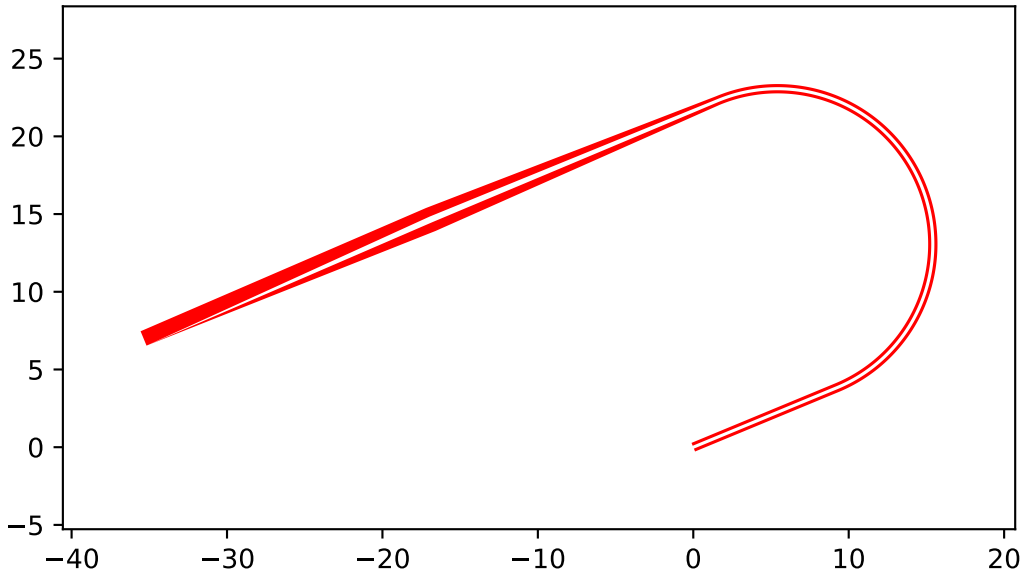
Alternatively, the width can also be an array for describing the dimensions of slot/coplanar waveguides. The array has the format *[rail_width_1, slot_width_1, rail_width_2, . . .]*, where the rail_widths describe the widths of the rails and the widths of the slots are defined by the slot_widths. This array can also end with a *slot_width*, which would lead to an asymmetry with respect to the center. This can e.g. be useful for tapering between single waveguides and slot waveguides.

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide

wg_1 = Waveguide.make_at_port(Port(origin=(0, 0), angle=np.pi / 8, width=[0.2,0.2,0.
↪2])) # array as width -> slot waveguide
```

(continues on next page)

(continued from previous page)

```
wg_1.add_straight_segment(10)
wg_1.add_bend(np.pi, 10)
wg_1.add_straight_segment(20, final_width=np.array([0.6,0.4,0.6])) # tapering between␣
↪two slot waveguides
wg_1.add_straight_segment(20, final_width=np.array([0,0,1])) # tapering to a ridge␣
↪waveguide

cell = Cell('CELL')
cell.add_to_layer(1, wg_1)  # red
cell.show()
```



3.3.2 Beam splitters

Beam splitters are used to split a beam in two parts. They are quite important for photonic circuits and are essential for interferometers. Three different types of beam splitters can be found in the gdshelpers library: Y-Splitters, Multimode Interferometers (MMIs) and Directional Couplers. Their corresponding classes *Splitter*, *MMI* and *DirectionalCoupler* can be found in the *splitter* package.

Y-Splitters

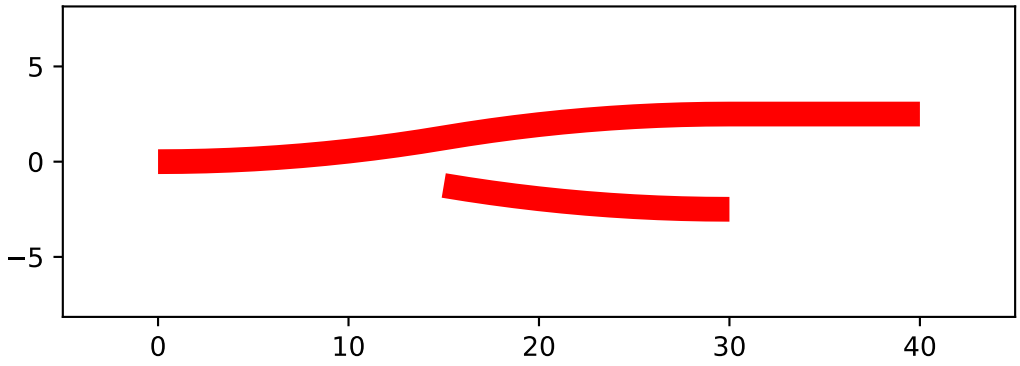
First, let us have a look at the Y-splitter. They have one input port and two output ports. If we want to create the splitter at an existing waveguide, we can use the functions *make_at_root_port*, *make_at_left_branch_port* or *make_at_right_branch_port*. To continue our waveguide from the splitter, we can address the left output port *left_branch_port* or the right output port *right_branch_port* of the splitter.

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.geometry import geometric_union
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.splitter import Splitter

splitter = Splitter(origin=(0, 0), angle=0, total_length=30, wg_width_root=1.3, sep=5)
```

(continues on next page)

(continued from previous page)

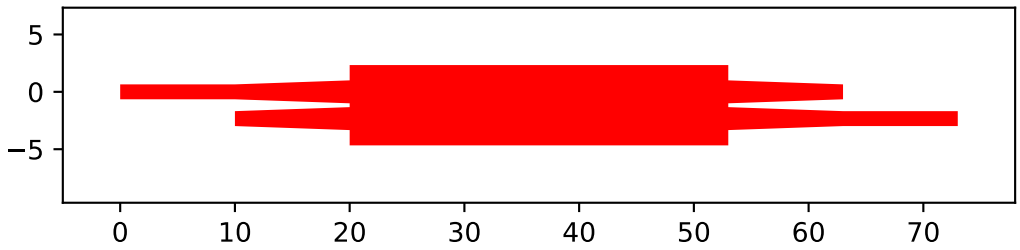
```
wg = Waveguide.make_at_port(splitter.left_branch_port)
wg.add_straight_segment(length=10)

cell = Cell('CELL')
cell.add_to_layer(1, splitter, wg)  # red
cell.show()
```



Multimode Interferometers

In contrast to the Y-splitters, MMIs can have an arbitrary number of input and output ports. However, typically only 2x2 or 1x2 MMIs are used. As before, it is a good idea to have a look at an example:

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts import Port
from gdshelpers.parts.splitter import MMI

waveguide_1 = Waveguide.make_at_port(Port((0, 0), 0, 1.3))
waveguide_1.add_straight_segment(length=10)
mmi = MMI.make_at_port(port=waveguide_1.current_port, length=33, width=7, num_
↪inputs=2, num_outputs=2, pos='i0')
waveguide_2 = Waveguide.make_at_port(mmi.output_ports[0])
waveguide_2.add_straight_segment(length=10)

cell = Cell('CELL')
cell.add_to_layer(1, waveguide_1, mmi, waveguide_2)
cell.show()
```



The first parameter of the the function `MMI.make_at_port()` defines the port of the MMI, meaning where the MMI is created and its rotation. The length and the width of the MMI are defined by the second and third parameter.

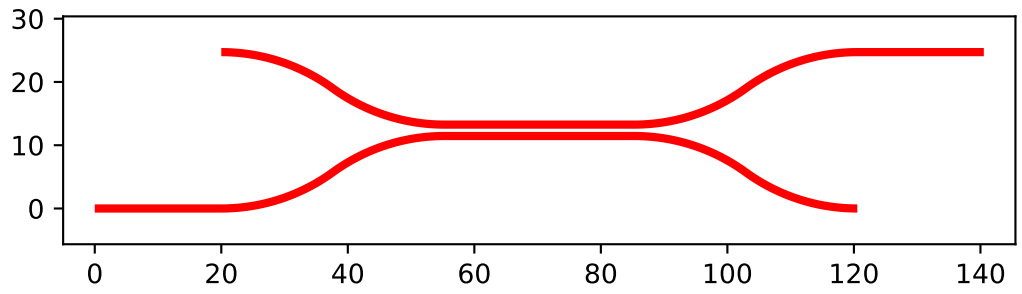
Choosing these parameters correctly is essential to achieve a good transmission and the desired splitting ration. Last but not least, we have to define the number of input and output ports, which are given by the last two parameters of the function. Apart from these parameters, the taper length and width can be optimized in order to increase the transmission. By default they are set to 10 um and 2 um.

As it can be seen, we have created a 2x2 MMI. At the moment, our input waveguide terminates in the upper input of the MMI. If we want it to terminate in the lower input, we have to change the position parameter to *i1*. Similarly, if we want to create a waveguide at the upper output, we have to replace *mmi.output_ports[0]* by *mmi.output_ports[1]*.

Directional Couplers

The last beam splitter we want to talk about is the directional coupler. It consists of two waveguides which are guided close to each other over a certain interaction length. As a consequence, we always have two input and two output ports. To make use of this coupler, we have to import the *DirectionalCoupler* class from the *splitter* library.

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts import Port
from gdshelpers.parts.splitter import DirectionalCoupler

waveguide_1 = Waveguide.make_at_port(port=Port((0, 0), angle=0, width=1.3))
waveguide_1.add_straight_segment(length=20)
DC = DirectionalCoupler.make_at_port(port=waveguide_1.current_port, length=30, gap=0.
↪5, bend_radius=30, which=0)
waveguide_2 = Waveguide.make_at_port(DC.right_ports[1])
waveguide_2.add_straight_segment(length=20)

cell = Cell('CELL')
cell.add_to_layer(1, waveguide_1, DC, waveguide_2)
cell.show()
```

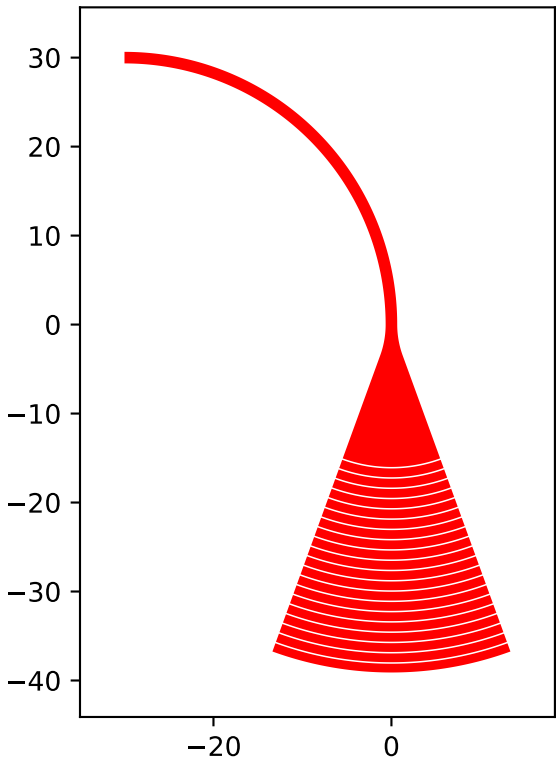


The origin of the coupler, the rotation and the width of the waveguides are tuned by the port parameter. Changing the width parameter we can decide whether we start at the lower (which = 0) or upper (which = 1)input. If we want our second waveguide to start at the other output of the coupler, we just have to replace *right_ports[1]* by *right_ports[0]*.

3.3.3 Grating Coupler

To couple light in our photonic circuits, grating coupler are of quite often the coupler of choice. Of course, these structures can be found in the gdshelpers library. To create such a coupler, first we have to import the *GratingCoupler* class. To start a waveguide from the coupler, we can make use of the *gc.port* parameter of the *GratingCoupler* class

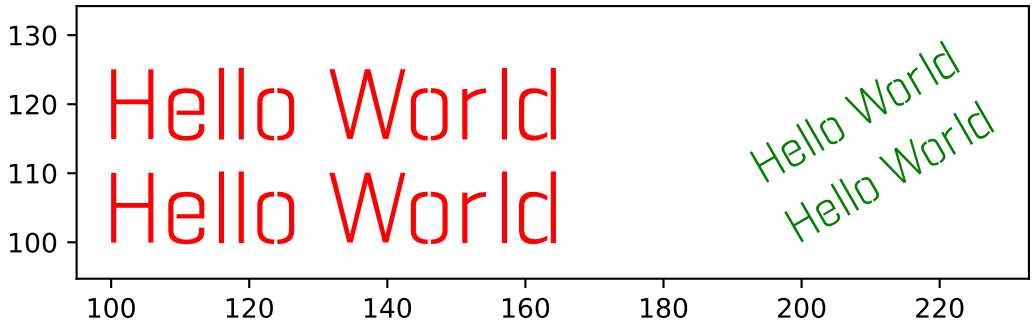
```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler

coupler_params = {
        'width': 1.3,
        'full_opening_angle': np.deg2rad(40),
        'grating_period': 1.155,
        'grating_ff': 0.85,
        'n_gratings': 20,
        'taper_length': 16.
}

gc = GratingCoupler.make_traditional_coupler(origin=(0, 0), **coupler_params)

wg = Waveguide.make_at_port(gc.port)
wg.add_bend(np.pi / 2, radius=30)

cell = Cell('CELL')
cell.add_to_layer(1, wg, gc)
cell.show()
```



3.3.4 Text

It is often quite useful to add text to the design, for example to identify a device under the microscope. Adding text is quite easy. We just have to import the class `Text()` from the package *gdshelpers.parts.text()*. The second step is to call the constructor of the class `Text()`.

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.text import Text

text_1 = Text(origin=[100, 100], height=10, text='Hello World\nHello World',␣
↪alignment='left-bottom')
text_2 = Text(origin=[200, 100], height=5, text='Hello World\nHello World', alignment=
↪'left-bottom', angle=np.pi / 6, line_spacing=2)

cell = Cell('CELL')
cell.add_to_layer(1, text_1) # red
cell.add_to_layer(2, text_2) # green
cell.show()
```



The first parameter *origin* denotes the position and the height of the text can be set by the second parameter *height*. The text itself can be given by the third parameter *text*. In addition, optional parameters as *alignment*, *angle* and *line_spacing* can be used the align, rotate the text and to vary the spacing between the lines.

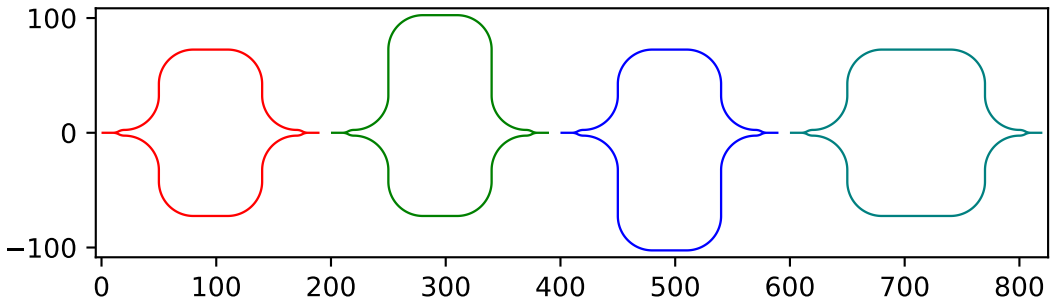
3.3.5 Mach Zehnder Interferometers

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.interferometer import MachZehnderInterferometer

wg_1 = Waveguide.make_at_port(Port((0, 0), angle=0, width=2))
wg_1.add_straight_segment(10)
mzi_1 = MachZehnderInterferometer.make_at_port(port=wg_1.current_port, splitter_
↪length=10, splitter_separation=5,

↪      bend_radius=30, upper_vertical_length=10, lower_vertical_length=10,

↪      horizontal_length=30)
wg_2 = Waveguide.make_at_port(mzi_1.port)
wg_2.add_straight_segment(10)

wg_3 = Waveguide.make_at_port(Port((200, 0), angle=0, width=2))
wg_3.add_straight_segment(10)
mzi_2 = MachZehnderInterferometer.make_at_port(port=wg_3.current_port, splitter_
↪length=10, splitter_separation=5,

↪      bend_radius=30, upper_vertical_length=40, lower_vertical_length=10,
```

(continues on next page)

(continued from previous page)

```
↪     horizontal_length=30)
wg_4 = Waveguide.make_at_port(mzi_2.port)
wg_4.add_straight_segment(10)

wg_5 = Waveguide.make_at_port(Port((400, 0), angle=0, width=2))
wg_5.add_straight_segment(10)
mzi_3 = MachZehnderInterferometer.make_at_port(port=wg_5.current_port, splitter_
↪length=10, splitter_separation=5,
↪     bend_radius=30, upper_vertical_length=10, lower_vertical_length=40,
↪     horizontal_length=30)
wg_6 = Waveguide.make_at_port(mzi_3.port)
wg_6.add_straight_segment(10)

wg_7 = Waveguide.make_at_port(Port((600, 0), angle=0, width=2))
wg_7.add_straight_segment(10)
mzi_4 = MachZehnderInterferometer.make_at_port(port=wg_7.current_port, splitter_
↪length=10, splitter_separation=5,
↪     bend_radius=30, upper_vertical_length=10, lower_vertical_length=10,
↪     horizontal_length=60)
wg_8 = Waveguide.make_at_port(mzi_4.port)
wg_8.add_straight_segment(10)

cell = Cell('CELL')
cell.add_to_layer(1, wg_1, wg_2, mzi_1)  # red
cell.add_to_layer(2, wg_3, wg_4, mzi_2)  # green
cell.add_to_layer(3, wg_5, wg_6, mzi_3)  # blue
cell.add_to_layer(4, wg_7, wg_8, mzi_4)  # teal
cell.show()
```



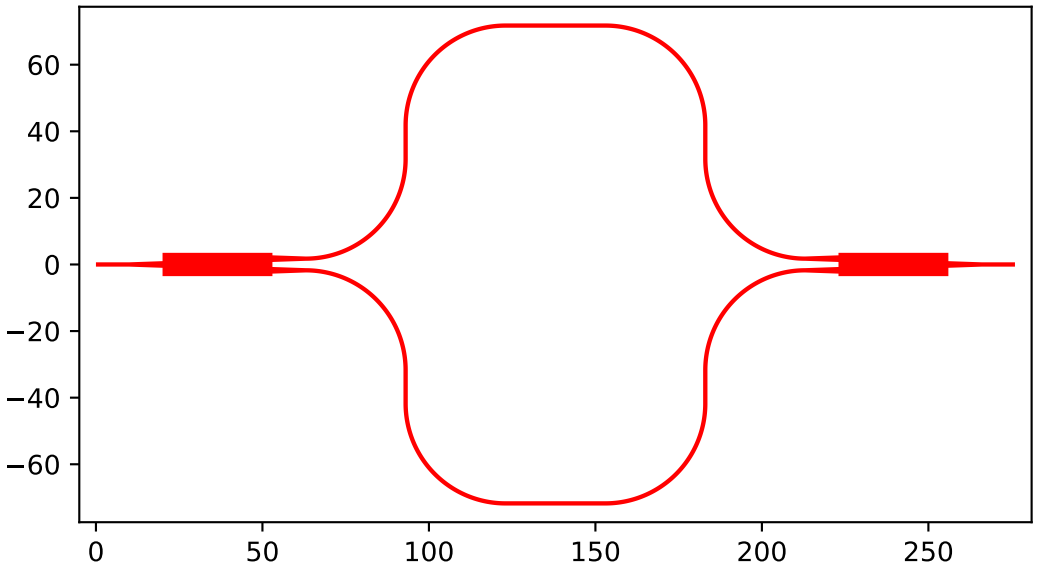
In this case a Y-Splitter was used. However, as MMIs are also frequently used, it makes sense to integrate them in this library. For this reason you can find a Mach Zehnder Interferometers

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.interferometer import MachZehnderInterferometerMMI

wg_1 = Waveguide.make_at_port(Port(origin=(0, 0), angle=0, width=1.3))
```

(continues on next page)

(continued from previous page)

```
wg_1.add_straight_segment(length=20)
mzi_1 = MachZehnderInterferometerMMI.make_at_port(port=wg_1.current_port, splitter_
↪length=33, splitter_width=7,
↪             bend_radius=30, upper_vertical_length=10, lower_vertical_length=10,
↪             horizontal_length=30)
wg_2 = Waveguide.make_at_port(port=mzi_1.port)
wg_2.add_straight_segment(length=20)

cell = Cell('CELL')
cell.add_to_layer(1, wg_1, wg_2, mzi_1)
cell.show()
```



3.3.6 Resonators

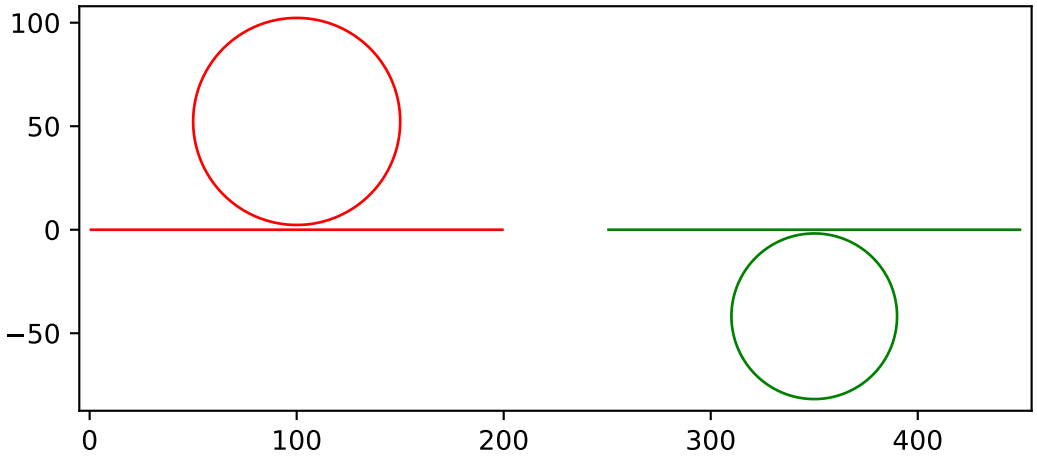
Ring Resonators

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.resonator import RingResonator
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide

waveguide_1 = Waveguide.make_at_port(Port([0, 0], 0, 1.3))
waveguide_1.add_straight_segment(100)
resonator_1 = RingResonator.make_at_port(waveguide_1.current_port, gap=1, radius=50)
waveguide_1.add_straight_segment(100)

waveguide_2 = Waveguide.make_at_port(Port([250, 0], 0, 1.3))
waveguide_2.add_straight_segment(100)
resonator_2 = RingResonator.make_at_port(waveguide_2.current_port, gap=-0.5,
↪radius=40)
waveguide_2.add_straight_segment(100)
```

(continues on next page)

(continued from previous page)

```
cell = Cell('CELL')
cell.add_to_layer(1, waveguide_1, resonator_1) # red
cell.add_to_layer(2, waveguide_2, resonator_2) # green
cell.show()
```



Apart from the port (origin, width and angle), the ring resonator is defined by the gap between waveguide and the ring as well as the radius of the ring.

3.3.7 Ports

Ports are constructs to make things easier. They are not visible in the final .gds file. Each port has three different properties:

- Origin
- Width
- Rotation

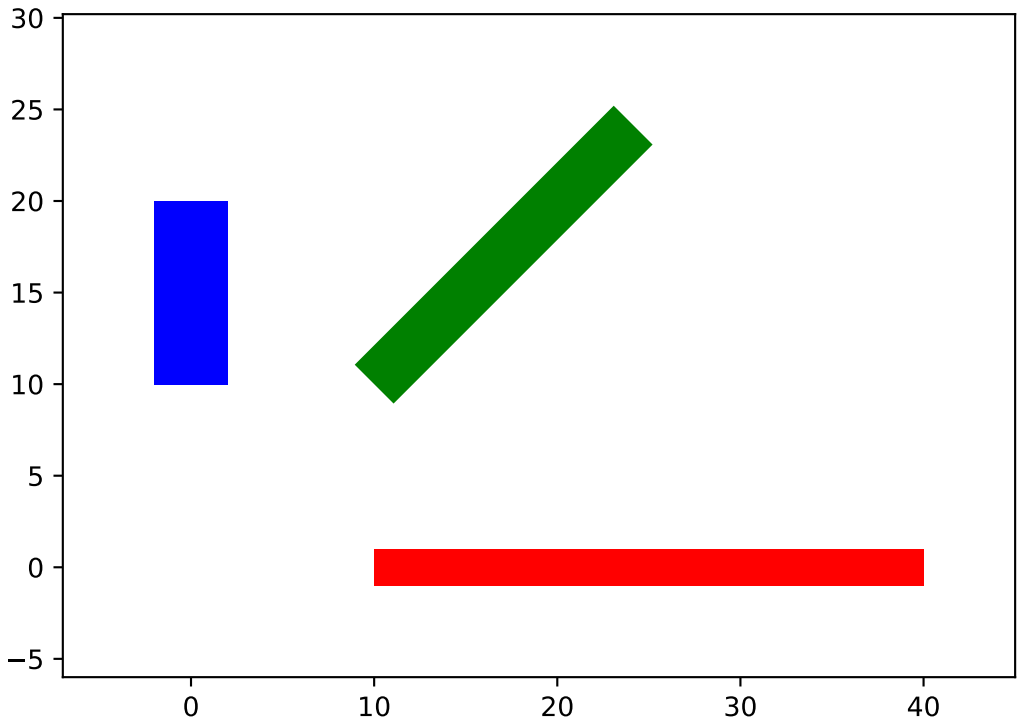
The width defines the width of all structures (e.g. waveguides) that start from this port. As always, the rotation is given in radiant and is calculated counterclockwise.

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.waveguide import Waveguide

wg_1 = Waveguide.make_at_port(Port(origin=[10, 0], angle=0, width=2))
wg_1.add_straight_segment(length=30)

wg_2 = Waveguide.make_at_port(Port(origin=[10, 10], angle=np.pi / 4, width=3))
wg_2.add_straight_segment(length=20)

wg_3 = Waveguide.make_at_port(Port(origin=[0, 10], angle=np.pi / 2, width=4))
wg_3.add_straight_segment(length=10)

cell = Cell('CELL')
```

(continues on next page)

(continued from previous page)

```
cell.add_to_layer(1, wg_1) # red
cell.add_to_layer(2, wg_2) # green
cell.add_to_layer(3, wg_3) # blue
cell.show()
```



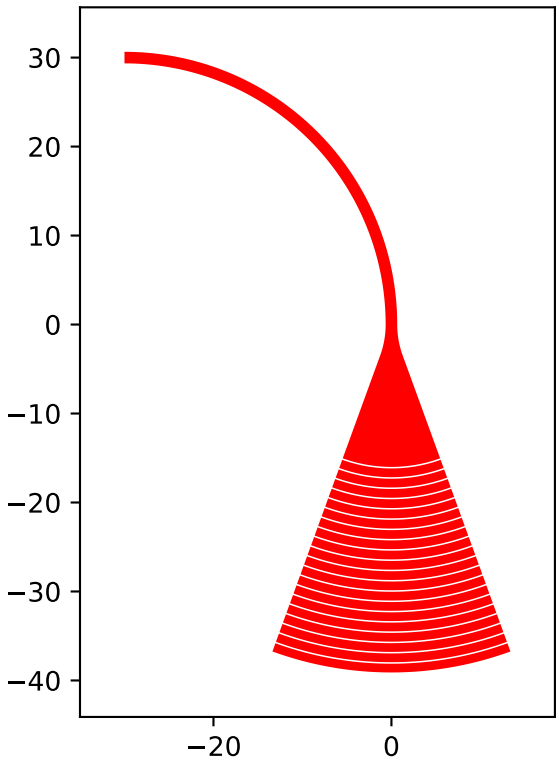
Many structures (e.g. waveguides, couplers, splitters) have a *port*, *current_port* or *output_ports* option. This can be used to start a new structure, e.g. a waveguide, from an old structure:

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler

coupler_params = {
        'width': 1.3,
        'full_opening_angle': np.deg2rad(40),
        'grating_period': 1.155,
        'grating_ff': 0.85,
        'n_gratings': 20,
        'taper_length': 16.
}

gc = GratingCoupler.make_traditional_coupler(origin=(0, 0), **coupler_params)

wg = Waveguide.make_at_port(gc.port)
wg.add_bend(np.pi / 2, radius=30)

cell = Cell('CELL')
cell.add_to_layer(1, wg, gc)
```

(continues on next page)

(continued from previous page)

```
cell.show()
```

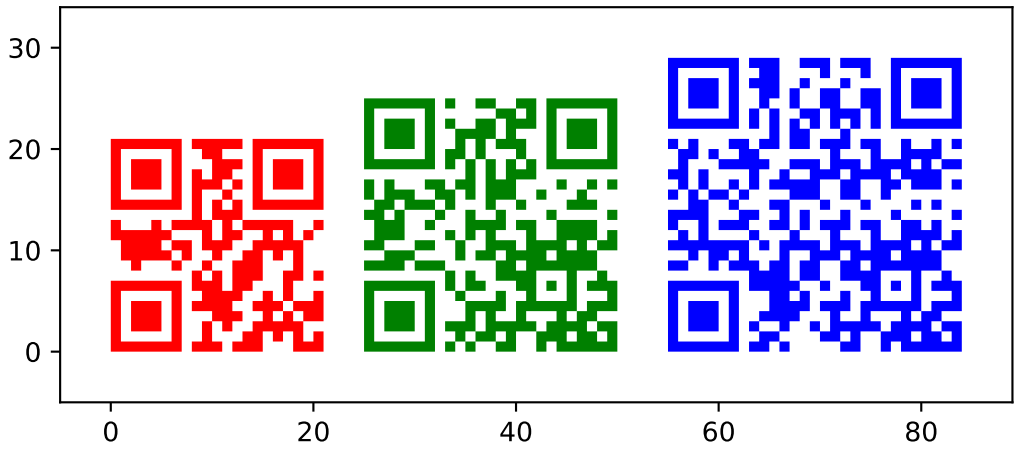


3.3.8 Optical Codes

QR Codes

Sometimes it can be useful to add QRCodes to a design. For example to create a link to a homepage. To add such a code, we have to import the *QRCode* class from the *optical_codes* package.

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.optical_codes import QRCode

qr_code_1 = QRCode(origin=[0, 0], data='A0.0', box_size=1.0, version=1, error_
↪correction=QRCode.ERROR_CORRECT_M)
qr_code_2 = QRCode(origin=[25, 0], data='A0.0', box_size=1.0, version=2, error_
↪correction=QRCode.ERROR_CORRECT_M)
qr_code_3 = QRCode(origin=[55, 0], data='A0.0', box_size=1.0, version=3, error_
↪correction=QRCode.ERROR_CORRECT_M)

cell = Cell('CELL')
cell.add_to_layer(1, qr_code_1) # red
cell.add_to_layer(2, qr_code_2) # green
cell.add_to_layer(3, qr_code_3) # blue
cell.show()
```

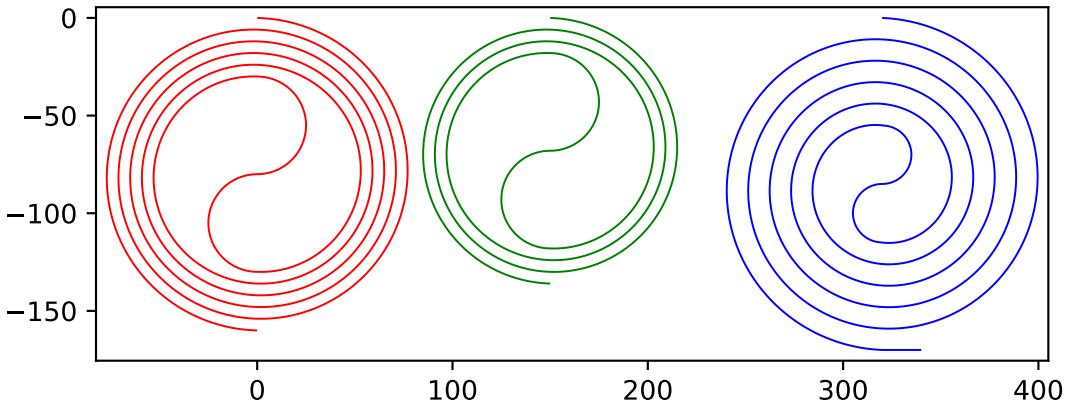


As always, the origin defines the point where the pattern is created. The data to be encoded is defined by the second parameter and the size of each element is defined by the *box_size* parameter.

3.3.9 Spiral

Sometimes long waveguides are needed, for example to determine the loss per length. Spirals are useful geometries for this case. For this purpose we can make use of the *Spiral* class, which can be found in the *spiral* package. A spiral is defined by four parameters: Port (origin, angle, width), number of turns, distance between two neighboring turns and the inner gap. At the end of the spiral we can continue our waveguide by referring to its output port *out_port*. Note that the spiral can not be defined over its length. However, we can readout the length with the *length* property.

```
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.port import Port
from gdshelpers.parts.spiral import Spiral
from gdshelpers.parts.waveguide import Waveguide

spiral_1 = Spiral.make_at_port(Port(origin=(0, 0), angle=0, width=1), num=5, gap=5,␣
↪inner_gap=50)
spiral_2 = Spiral.make_at_port(Port(origin=(150, 0), angle=0, width=1), num=3, gap=5,␣
↪inner_gap=50)
spiral_3 = Spiral.make_at_port(Port(origin=(320, 0), angle=0, width=1), num=5, gap=10,
↪ inner_gap=30)
length = spiral_3.length
wg = Waveguide.make_at_port(spiral_3.out_port)
wg.add_straight_segment(20)

cell = Cell('Spiral')
cell.add_to_layer(1, spiral_1)  # red
cell.add_to_layer(2, spiral_2)  # green
cell.add_to_layer(3, spiral_3, wg)  # blue
cell.show()
```



3.4 Modifier

3.4.1 From negative layout to postive layout

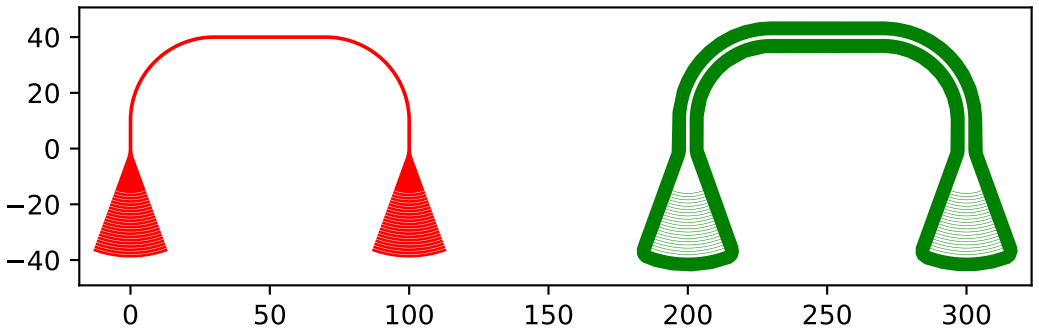
Sometimes, using positive resist to pattern the waveguides gives better results compared to negative resist. To make this way of designing the structures as easy as possible, we implemented a function called `convert_to_positve_resist()`. Apart from the defining the structure itself, the only parameter that has to be given is the buffer radius. Here is a short example of two photonic routings, one patterned with negative (red) and one patterned with positive (green) resist:

```
import numpy as np
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.coupler import GratingCoupler
from gdshelpers.helpers.positive_resist import convert_to_positive_resist

coupler_params = {
    'width': 1.3,
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

# negative resist
left_coupler_1 = GratingCoupler.make_traditional_coupler((0,0), **coupler_params)
right_coupler_1 = GratingCoupler.make_traditional_coupler((100,0), **coupler_params)

wg_1 = Waveguide.make_at_port(left_coupler_1.port)
wg_1.add_straight_segment(10)
wg_1.add_bend(-np.pi / 2, 30)
wg_1.add_straight_segment_until_x(right_coupler_1.port.x - 30)
wg_1.add_bend(-np.pi / 2, 30)
wg_1.add_straight_segment(10)

# positive resist
left_coupler_2 = GratingCoupler.make_traditional_coupler((200,0), **coupler_params)
right_coupler_2 = GratingCoupler.make_traditional_coupler((300,0), **coupler_params)
```

(continues on next page)

(continued from previous page)

```
wg_2 = Waveguide.make_at_port(left_coupler_2.port)
wg_2.add_straight_segment(10)
wg_2.add_bend(-np.pi / 2, 30)
wg_2.add_straight_segment_until_x(right_coupler_2.port.x - 30)
wg_2.add_bend(-np.pi / 2, 30)
wg_2.add_straight_segment(10)

cell = Cell('SIMPLE_DEVICE')
cell.add_to_layer(1, left_coupler_1, wg_1, right_coupler_1)
cell.add_to_layer(2, convert_to_positive_resist(parts=[wg_2, left_coupler_2, right_
↪coupler_2], buffer_radius=5))
cell.show()
```



As it can be seen, the workflow is exactly the same as for negative resist and only one additional function has to be added. Of course, this does also work for more complex designs.

3.4.2 Creating holes for underetching

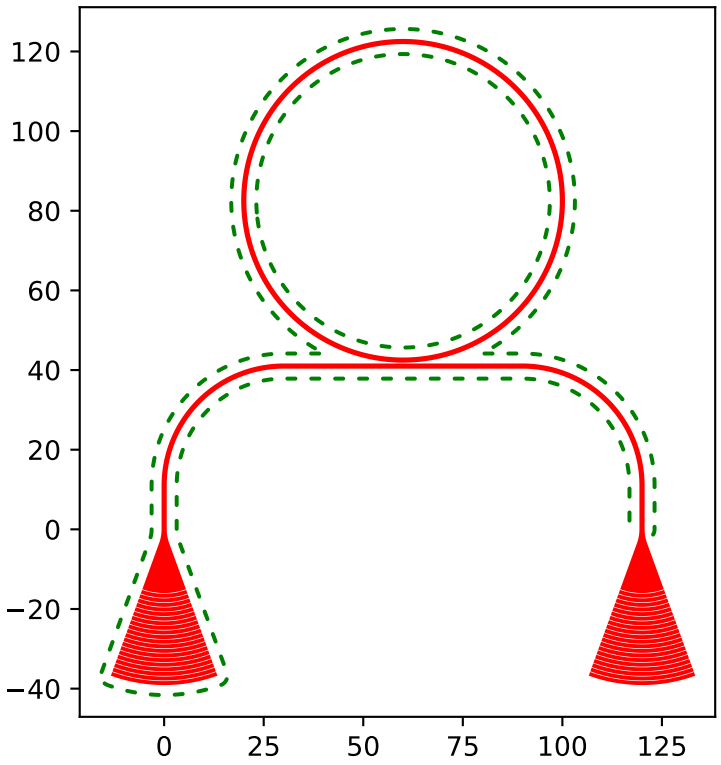
To creates holes around defined parts, which can be used for underetching processes, we implemented a `create_holes_for_under_etching()` function. For this example, let us consider a grating coupler, a waveguide, a ring resonator and a second grating coupler. First, we have to define the parts which shall be underetched, in this case the left grating coupler, waveguide and ring resonator. If the complete structure is underetched, then you will not notice any problems. However, if one part of your structure is not underetched, for example the right grating coupler, then you might get a collision between the photonics layer and the hole layer. For this reason, in addition to the *underetching_parts*, we have to define the *complete_structure*, which is used to prevent overlapping. While the *underetching_parts* contains the left grating coupler, waveguide and resonator, the *complete_structure* contains the *underetching_parts* and additonally, the second grating coupler. Finally, the radius, distance, spacing and and length of the holes ca be adjusted using the corresponding parameters.

```
import numpy as np
from gdshelpers.geometry import geometric_union
from gdshelpers.parts.coupler import GratingCoupler
from gdshelpers.geometry.chip import Cell
from gdshelpers.parts.waveguide import Waveguide
from gdshelpers.parts.resonator import RingResonator
from gdshelpers.helpers.under_etching import create_holes_for_under_etching

coupler_params = {
    'width': 1.3,
```

(continues on next page)

(continued from previous page)

```
    'full_opening_angle': np.deg2rad(40),
    'grating_period': 1.155,
    'grating_ff': 0.85,
    'n_gratings': 20,
    'taper_length': 16.
}

# ==== create some sample structures (straight line with ring resonator)
coupler_1 = GratingCoupler.make_traditional_coupler((0,0), **coupler_params)
wg_1 = Waveguide.make_at_port(coupler_1.port)
wg_1.add_straight_segment(11)
wg_1.add_bend(-np.pi / 2, 30)
wg_1.add_straight_segment(30)
resonator = RingResonator.make_at_port(port=wg_1.current_port, gap=0.2, radius=40)
wg_1.add_straight_segment(30)
wg_1.add_bend(-np.pi / 2, 30)
wg_1.add_straight_segment(11)
coupler_2 = GratingCoupler.make_traditional_coupler(wg_1.current_port.origin,␣
↪**coupler_params)

underetching_parts = geometric_union([wg_1, resonator, coupler_1])
complete_structure = geometric_union([underetching_parts, coupler_2])
# create the holes with a radius of 0.5 microns, a distance of 2 microns to the␣
↪structure borders and
# a distance of 2 microns between the holes
holes = create_holes_for_under_etching(underetch_parts=underetching_parts, complete_
↪structure=complete_structure,
                                       hole_radius=0.5, hole_distance=2, hole_
↪spacing=3, hole_length=3)

# create a cell with the structures in layer 1 and the holes in layer 2
cell = Cell('CELL')
cell.add_to_layer(1, complete_structure)
cell.add_to_layer(2, holes)
cell.show()
```



3.5 gdshelpers

3.5.1 gdshelpers package

Subpackages

gdshelpers.export package

Submodules

gdshelpers.export.blender_export module

class gdshelpers.export.blender_export.**Shapely3d**(*poly*, *extrude_val_min*, *extrude_val_max*, *rgb=(255, 255, 255)*)

Bases: object

This class saves the values which will be given to blender and it also provides a method for output these values

to_temp(*scale*, *index*)

gdshelpers.export.blender_export.**render_image**(*shapely_3d_list*, *filename*, *scale=0.1*, *resolution_x=1980*, *resolution_y=1080*, *resolution_percentage=100*, *render_engine='CYCLES'*, *camera_position_y='above'*, *camera_position_x='right'*)

writes the 'filename.png' file

Parameters

- **shapely_3d_list** – a list of the meshes to export
- **filename** – the name of the .png and .tmp file
- **scale** – size factor
- **resolution_x** – set x resolution for a later render process
- **resolution_y** – set y resolution for a later render process
- **resolution_percentage** – set resolution percentage
- **render_engine** – set the render engine
- **camera_position_y** – decides if the camera is placed 'above' or 'under' the device
- **camera_position_x** – decides if the camera is places 'right' or 'left' of the device

Returns nothing

gdshelpers.export.blender_export.**render_image_and_save_as_blend**(*shapely_3d_list*, *filename*, *scale=0.1*, *resolution_x=1980*, *resolution_y=1080*, *resolution_percentage=100*, *render_engine='CYCLES'*, *camera_position_y='above'*, *camera_position_x='right'*)

writes the 'filename.png' and 'filename.blend' file

Parameters

- **shapely_3d_list** – a list of the meshes to export
- **filename** – the name of the .png, .blend and .tmp file
- **scale** – size factor
- **resolution_x** – set x resolution for a later render process
- **resolution_y** – set y resolution for a later render process
- **resolution_percentage** – set resolution percentage
- **render_engine** – set the render engine
- **camera_position_y** – decides if the camera is placed 'above' or 'under' the device
- **camera_position_x** – decides if the camera is places 'right' or 'left' of the device

Returns nothing

gdshelpers.export.blender_export.**save_as_blend**(*shapely_3d_list*, *filename*, *scale=0.1*, *resolution_x=1980*, *resolution_y=1080*, *resolution_percentage=100*, *render_engine='CYCLES'*, *camera_position_y='above'*, *camera_position_x='right'*)

writes the 'filename.blend' file

Parameters

- **shapely_3d_list** – a list of the meshes to export
- **filename** – the name of the .blend and .tmp file
- **scale** – size factor
- **resolution_x** – set x resolution for a later render process
- **resolution_y** – set y resolution for a later render process
- **resolution_percentage** – set resolution percentage
- **render_engine** – set the render engine
- **camera_position_y** – decides if the camera is placed 'above' or 'under' the device
- **camera_position_x** – decides if the camera is places 'right' or 'left' of the device

Returns nothing

gdshelpers.export.dxf_export module

gdshelpers.export.gdsii_export module

For documentation of GDSII see: http://bitsavers.informatik.uni-stuttgart.de/pdf/calma/GDS_II_Stream_Format_Manual_6.0_Feb87.pdf

gdshelpers.export.gdsii_export.**write_cell_to_gdsii_file**(*outfile*, *cell*, *unit=1e-06*, *grid_steps_per_unit=1000*, *max_points=4000*, *max_line_points=4000*, *timestamp=None*, *parallel=False*, *max_workers=None*)

Module contents

gdshelpers.geometry package

Submodules

gdshelpers.geometry.chip module

class gdshelpers.geometry.chip.**Cell**(*name: str*)

Bases: object

add_cell(*cell*, *origin=(0, 0)*, *angle: Optional[float] = None*, *columns=1*, *rows=1*, *spacing=None*)
Adds a Cell to this cell

Parameters

- **cell** – Cell to add
- **origin** – position where to add the cell
- **angle** – defines the rotation of the cell
- **columns** – Number of columns
- **rows** – Number of rows
- **spacing** – Spacing between the cells, should be an array in the form [x_spacing, y_spacing]

add_dlw_data(*dlw_type*, *dlw_id*, *data*)
Adds data for 3D-hybrid-integration to the Cell. This is usually only done by using the Device

Parameters

- **dlw_type** – type of the represented object
- **dlw_id** – id of the represented object
- **data** – data of the object

add_dlw_marker(*label: str*, *layer: int*, *origin*, *box_size=2.5*)
Adds a marker for 3D-hybrid integration

Parameters

- **label** – Name of the marker, needs to be unique within the device
- **layer** – Layer at which the marker and markers should be written
- **origin** – Position of the marker
- **box_size** – Size of the box of the marker

add_dlw_taper_at_port(*label: str*, *layer: int*, *port: gdshelpers.parts.port.Port*, *taper_length: float*, *tip_width=0.01*, *with_markers=True*, *box_size=2.5*)
Adds a taper for 3D-hybrid-integration at a certain port

Parameters

- **label** – Name of the port, needs to be unique within the device
- **layer** – Layer at which the taper and markers should be written
- **port** – Port to which the taper should be attached
- **taper_length** – length of the taper
- **tip_width** – final width of the tip
- **with_markers** – for recognizing the taper markers near to the taper are necessary. In certain designs the standard positions are not appropriate and can therefore be disabled and manually added
- **box_size** – Size of the box of the markers

add_ebl_frame(*layer: int*, *frame_generator*, *bounds=None*, ***kwargs*)
Adds global markers to the layout

Parameters

- **layer** – layer on which the markers should be positioned
- **frame_generator** – either a method, which returns a list of the markers, which should be added or the name of a generator from the gdshelpers.geometry.ebl_frame_generators package
- **bounds** – Optionally the bounds to use can be provided in the form (min_x, min_y, max_x, max_y). If None, the standard cell bounds will be used.
- **kwargs** – Parameters which are directly passed to the frame generator (other than the bounds parameter)

add_ebl_marker(*layer: int*, *marker*)
Adds an Marker to the layout

Parameters

- **layer** – layer on which the marker should be positioned
- **marker** – marker, that should be added (from gdshelpers.parts.markers)

add_frame(*padding=30.0*, *line_width=1.0*, *frame_layer: int = 6*, *bounds=None*)
Generates a rectangular frame around the contents of the cell.

Parameters

- **padding** – Add a padding of the given value around the contents of the cell
- **line_width** – Width of the frame line
- **frame_layer** – Layer to put the frame on.
- **bounds** – Optionally, an explicit extent in the form (min_x, min_y, max_x, max_y) can be passed to the function. If *None* (default), the current extent of the cell will be chosen.

add_region_layer(*region_layer: int = 5*, *layers: Optional[List[int]] = None*)
Generate a region layer around all objects on *layers* and place it on layer *region_layer*. If *layers* is None, all layers are used.

add_to_desc(*key*, *data*)
Adds data to the .desc-file can be used for any data related to the cells, e.g. the swept parameters/...

Parameters

- **key** – name of the entry
- **data** – data which are added to the .desc-file, the data have to be serializable by json

add_to_layer(*layer: int*, **geometry*)
Adds a shapely geometry to a the layer

Parameters

- **layer** – id of the layer, a tuple (layer, datatype) can also be passed to define the datatype as well
- **geometry** – shapely geometry

bounds
The outer bounding box of the cell. Returns *None* if it is empty.

export_mesh(*filename: str*, *layer_defs*)
Saves the current geometry as a mesh-file.

Parameters

- **filename** – Name of the file which will be created. The file ending determines the format.
- **layer_defs** – Definition of the layers, should be a list like [(layer,(z_min,z_max)),...]

get_bounds(*layers: Optional[List[int]] = None*)
Calculates and returns the envelope for the given layers. Returns *None* if it is empty.

get_desc()

get_dlw_data()

get_fractured_layer_dict(*max_points=4000*, *max_line_points=4000*)

get_gdspy_cell(*executor=None*)

get_gdspy_lib()

get_oasis_cells(*grid_steps_per_micron=1000*, *executor=None*)

get_patches(*origin=(0, 0)*, *angle_sum=0*, *angle=0*, *layers: Optional[List[int]] = None*)

get_reduced_layer(*layer: int*)
Returns a single shapely object containing the structures on a certain layer from this cell and all added cells.

Parameters **layer** – the layer whose structures will be returned

Returns a single shapely-geometry

save(*name=None*, *library=None*, *grid_steps_per_micron=1000*, *parallel=False*, *max_workers=None*)
Exports the layout and creates an DLW-file, if DLW-features are used.

Parameters

- **name** – The filename of the saved file. The ending of the filename defines the format. Currently .gds, .oasis and .dxf are supported.
- **library** – Name of the used library. Should stay *None* in order to select the library depending on the file-ending. The use of this parameter is deprecated and this parameter will be removed in a future release.
- **grid_steps_per_micron** – Defines the resolution
- **parallel** – Defines if parallelization is used (only supported in Python 3). Standard value will be changed to True in a future version. Deactivating can be useful for debugging reasons.
- **max_workers** – If parallel is True, this can be used to limit the number of parallel processes. This can be useful if you run into out-of-memory errors otherwise.

save_desc(*filename: str*)
Saves a description file for the layout. The file format is not final yet and might change in a future release.

Parameters **filename** – name of the file the description data will be written to

save_image(*filename: str*, *layers: Optional[List[int]] = None*, *antialiased=True*, *resolution=1.0*, *ylim=(None, None)*, *xlim=(None, None)*, *scale=1.0*)
Save cell object as an image.

You can either use a rasterized file format such as png but also formats such as SVG or PDF.

Parameters

- **filename** – Name of the image file.
- **layers** – Layers to show im the image

- **resolution** – Rasterization resolution in GDSII units.
- **antialiased** – Whether to use a anti-aliasing or not.
- **ylim** – Tuple of (min_x, max_x) to export.
- **xlim** – Tuple of (min_y, max_y) to export.
- **scale** – Defines the scale of the image

show(*layers: Optional[List[int]] = None*, *padding=5*)
Shows the current cell

Parameters

- **layers** – List of the layers to be shown, passing None shows all layers
- **padding** – padding around the structure

size
Returns the size of the cell

start_viewer()

gdshelpers.geometry.ebl_frame_generators module

gdshelpers.geometry.ebl_frame_generators.**raith_marker_frame**(*bounds*, *padding=100*, *pitch=200*, *size=20*, *n=5*)
Generates a list of markers markers in each corner around the given bounding box. In each corner the markers are arranged in an L shape. This allows to have more markers (for exposure of many steps sometimes more than four markers are necessary) with larger distance between the markers (minimize risk of wrong markers being found by the EBL) without taking up too much space.

Parameters

- **bounds** – The bounds around which the markers will be arranged (the marker centers will be inside these bounds)
- **padding** – Spacing between the given bounds and the markers. Can also be negative to place the markers inside the bounding box.
- **pitch** – Pitch between two adjacent markers
- **size** – The marker size
- **n** – This determines the number of markers: There will be (2*n)+1 markers in each corner.

gdshelpers.geometry.shapely_adapter module

gdshelpers.geometry.shapely_adapter.**bounds_union**(*bound_list*)
Calculates the bounding box of all bounding boxes in the given list. Each bbox has to be given as (xmin, ymin, xmax, ymax) tuple.

Parameters **bound_list** – List of tuples containing all bounding boxes to be merged

gdshelpers.geometry.shapely_adapter.**convert_to_gdscad**(*objs*, *layer=1*, *datatype=None*, *path_width=1.0*, *path_pathtype=0*, *max_points=None*, *over_fracture_factor=1*, *max_points_line=None*)

Convert any shapely or list of shapely objects to a list of gdsCAD objects.

Since Shapely objects do not contain layer information nor line width for Shapely lines, you have to specify these during conversion. Typically, you will only convert polygons and specify the *layer*.

Export options such as *datatype* and maximum number of points per Polygon/Line default to the current module default.

On special feature is the *over_fracture_factor*. The polygons will be fractured into *max_points/over_fracture_factor* points and then healed again - which results in better fragmentation of some geometries. An *over_fracture_factor* of 0 will suppress healing.

Parameters

- **objs** – Part or Shapely object or list of Parts and/or Shapely objects.
- **layer** (*int*) – Layer on which to put the objects.
- **datatype** (*int*) – GDS datatype of the converted objects. Defaults to module wide settings.
- **path_width** (*float*) – With of GDS path, converted from Shapely lines.
- **path_pathtype** (*int*) – GDS path end type.
- **max_points** (*int, None*) – Maximum number of points. Defaults to module wide settings.
- **max_points_line** (*int, None*) – Maximum number of points for lines. Defaults to module wide settings.
- **over_fracture_factor** (*int*) – Break polygons in *over_fracture_factor* times smaller objects first. Then merge them again. May result in better fracturing but high numbers increase conversion time.

  When *over_fracture_factor* is set to 0, no additional healing is done.

gdshelpers.geometry.shapely_adapter.**convert_to_layout_objs**(*objs*, *layer=1*, *datatype=None*, *path_width=1.0*, *path_pathtype=0*, *max_points=None*, *over_fracture_factor=1*, *max_points_line=None*, *library='gdscad'*, *grid_steps_per_micron=1000*)

Convert any shapely or list of shapely objects to a list of gdsCAD objects.

Since Shapely objects do not contain layer information nor line width for Shapely lines, you have to specify these during conversion. Typically, you will only convert polygons and specify the *layer*.

Export options such as *datatype* and maximum number of points per Polygon/Line default to the current module default.

On special feature is the *over_fracture_factor*. The polygons will be fractured into *max_points*/*over_fracture_factor* points and then healed again - which results in better fragmentation of some geometries. An *over_fracture_factor* of 0 will suppress healing.

Parameters

- **objs** – Part or Shapely object or list of Parts and/or Shapely objects.
- **layer** (*int*) – Layer on which to put the objects.
- **datatype** (*int*) – GDS datatype of the converted objects. Defaults to module wide settings.
- **path_width** (*float*) – With of GDS path, converted from Shapely lines.
- **path_pathtype** (*int*) – GDS path end type.
- **max_points** (*int, None*) – Maximum number of points. Defaults to module wide settings.
- **max_points_line** (*int, None*) – Maximum number of points for lines. Defaults to module wide settings.
- **over_fracture_factor** (*int*) – Break polygons in *over_fracture_factor* times smaller objects first. Then merge them again. May result in better fracturing but high numbers increase conversion time.

  When *over_fracture_factor* is set to 0, no additional healing is done.
- **library** (*str*) – Defines the used library, either gdscad or gdspy
- **grid_steps_per_micron** (*int*) – Number of steps of the grid per micron, defaults to 1000 steps per micron

gdshelpers.geometry.shapely_adapter.**cut_shapely_object**(*obj*, *x_axis=None*, *y_axis=None*, *other_side=False*)

Cut a shapely object into two halves.

If both x_axis and y_axis are None, they are assumed to be the center of the bounding box.

If only one of the axis is given, and the other axis is None, the object is cut at the given axis.

If both values are given, or inferred to be the center of the bounding box due to being both None, the object is cut along its longer side.

Parameters

- **obj** (*shapely.base.BaseGeometry*) – Basically any shapely object
- **x_axis** (*float, None*) – x-axis cut value.
- **y_axis** – y-axis cut value.
- **other_side** – If the cut axis is determined automatically, use the other direction.

Returns A tuple of two shapely objects.

Return type list

gdshelpers.geometry.shapely_adapter.**fracture**(*obj*, *max_points_poly*, *max_points_line*, *max_interior=0*)

gdshelpers.geometry.shapely_adapter.**fracture_intelligently**(*obj*, *max_points*, *max_points_line*, *over_fracture_factor=1*)

gdshelpers.geometry.shapely_adapter.**geometric_union**(*objs*)
Join a list of Parts and/or Shapely objects to one big Shapely object.

Parameters **objs** – List of Parts and Shapely objects.

Returns Merged Shapely geometry.

Return type shapely.base.BaseGeometry

gdshelpers.geometry.shapely_adapter.**heal**(*objs*, *max_points*, *max_interior=0*)
Heal a list of Shapely geometries.

All elements touching each other will be merged as long as the number of points remains below *max_points*.

Parameters

- **objs** (*list, tuple*) – List Shapely geometries.
- **max_points** (*int*) – Maximum number of points.

Returns List of healed Shapely geometries.

Return type list

gdshelpers.geometry.shapely_adapter.**shapely_collection_to_basic_objs**(*collection*)
Convert the object or the collection to a list of basic shapely objects.

Parameters **collection** –

Returns

rtype

gdshelpers.geometry.shapely_adapter.**transform_bounds**(*bounds*, *origin*, *rotation=0*, *scale=1.0*)
Transform a bounds tuple (xmin, ymin, xmax, ymax) by the given offset, rotation and scale.

Module contents

gdshelpers.geometry.**convert_to_gdscad**(*objs*, *layer=1*, *datatype=None*, *path_width=1.0*, *path_pathtype=0*, *max_points=None*, *over_fracture_factor=1*, *max_points_line=None*)
Convert any shapely or list of shapely objects to a list of gdsCAD objects.

Since Shapely objects do not contain layer information nor line width for Shapely lines, you have to specify these during conversion. Typically, you will only convert polygons and specify the *layer*.

Export options such as *datatype* and maximum number of points per Polygon/Line default to the current module default.

On special feature is the *over_fracture_factor*. The polygons will be fractured into *max_points*/*over_fracture_factor* points and then healed again - which results in better fragmentation of some geometries. An *over_fracture_factor* of 0 will suppress healing.

Parameters

- **objs** – Part or Shapely object or list of Parts and/or Shapely objects.
- **layer** (*int*) – Layer on which to put the objects.
- **datatype** (*int*) – GDS datatype of the converted objects. Defaults to module wide settings.
- **path_width** (*float*) – With of GDS path, converted from Shapely lines.

- **path_pathtype** (*int*) – GDS path end type.
- **max_points** (*int, None*) – Maximum number of points. Defaults to module wide settings.
- **max_points_line** (*int, None*) – Maximum number of points for lines. Defaults to module wide settings.
- **over_fracture_factor** (*int*) – Break polygons in *over_fracture_factor* times smaller objects first. Then merge them again. May result in better fracturing but high numbers increase conversion time.

  When *over_fracture_factor* is set to 0, no additional healing is done.

gdshelpers.geometry.**geometric_union**(*objs*)

Join a list of Parts and/or Shapely objects to one big Shapely object.

Parameters **objs** – List of Parts and Shapely objects.

Returns Merged Shapely geometry.

Return type shapely.base.BaseGeometry

gdshelpers.geometry.**cut_shapely_object**(*obj*, *x_axis=None*, *y_axis=None*, *other_side=False*)

Cut a shapely object into two halves.

If both x_axis and y_axis are None, they are assumed to be the center of the bounding box.

If only one of the axis is given, and the other axis is None, the object is cut at the given axis.

If both values are given, or inferred to be the center of the bounding box due to being both None, the object is cut along its longer side.

Parameters

- **obj** (*shapely.base.BaseGeometry*) – Basically any shapely object
- **x_axis** (*float, None*) – x-axis cut value.
- **y_axis** – y-axis cut value.
- **other_side** – If the cut axis is determined automatically, use the other direction.

Returns A tuple of two shapely objects.

Return type list

gdshelpers.geometry.**fracture**(*obj*, *max_points_poly*, *max_points_line*, *max_interior=0*)

gdshelpers.helpers package

Submodules

gdshelpers.helpers.alignment module

class gdshelpers.helpers.alignment.**Alignment**(*alignment='left-bottom'*)

Bases: object

Alignment helper class.

Given a bounding box in the form of `((min_x, min_y), (max_x, max_y))`, this class returns special points of this box:

```
(max_y)     top +------+------+
                |      |      |
                |      |      |
     center     +------+------+
                |      |      |
                |      |      |
(min_y) bottom +------+------+
           (min_x)          (max_x)
             Left   Center    Right
```

Alignment options are given as - separated tuple, allowing for combinations of `left`, `center`, `right` with `bottom`, `center`, `top`.

alignment
Property holding the current alignment.

Returns Alignment string, i.e. `'bottom-left'`.

Return type str

alignment_functions
Returns a 2-tuple of functions, calculating the offset coordinates for a given bounding box.

Returns Tuple of functions.

Return type tuple

calculate_offset(*bbox*)
Calculate the coordinates of the current alignment for the given bounding box *bbox*.

Parameters **bbox** – Bounding box in the `((min_x, min_y), (max_x, max_y))` format.

Returns (x, y) offset coordinates.

Return type np.array

gdshelpers.helpers.bezier module

class gdshelpers.helpers.bezier.**CubicBezierCurve**(*p0*, *p1*, *p2*, *p3*)
Bases: `object`

evaluate(*t*)

evaluate_d1(*t*)

split(*t*)
Split the cubic Bezier curve into two new cubic Bezier, both describing one part of the curve.

Parameters **t** –

Returns

gdshelpers.helpers.layers module

gdshelpers.helpers.layers.**devnamelayer = 8**
Layer for device names. (For example A1, F0, etc..)

gdshelpers.helpers.layers.**framelayer = 6**
Layer for frames. (For overall pattern and/or arrays of devices)

gdshelpers.helpers.layers.**gmarklayer = 2**
Layer for global markers (Crosses, . . .)

gdshelpers.helpers.layers.**gplayers = (16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 2**
List of unassigned layers for general purpose use.

gdshelpers.helpers.layers.**lmarklayer = 3**
Layer for local markers (Crosses, . . .)

gdshelpers.helpers.layers.**masklayer1 = 14**
Mask window layer 1. (opening windows for subsequent etching e.g. freestanding devices)

gdshelpers.helpers.layers.**masklayer2 = 15**
Mask window layer 2. (opening windows for subsequent etching e.g. freestanding devices)

gdshelpers.helpers.layers.**nanolayer = 13**
Nano features layer. (SSPD wires, high resolution elements. . .)

gdshelpers.helpers.layers.**outlayer = 1**
Layer for photonic devices (Waveguides, Couplers, Splitters. . .)

gdshelpers.helpers.layers.**padlayer = 11**
Metal contact pad layer. (For probe contacts or wire bonding)

gdshelpers.helpers.layers.**parnamelayer1 = 9**
Layer for local parameters. (Period of couplers, gaps, . . . Used for first lithography step)

gdshelpers.helpers.layers.**parnamelayer2 = 10**
Layer for local parameters. (Period of couplers, gaps, . . . Used for second lithography step)

gdshelpers.helpers.layers.**patnamelayer = 7**
Layer for pattern names. (For example NG01 CalibCoupler)

gdshelpers.helpers.layers.**regionlayer = 5**
Layer for region definition. (For defining working areas = multiple of writefield_size)

gdshelpers.helpers.layers.**wflayer = 4**
Layer for write fields. (Used just for checking, not used in lithography)

gdshelpers.helpers.layers.**winglayer = 12**
Wing layer. (For connection between nanowires and pads)

gdshelpers.helpers.positive_resist module

gdshelpers.helpers.positive_resist.**convert_to_positive_resist**(*parts*, *buffer_radius*, *outer_resolution=None*, *clearance_features=None*, *exclude=None*)
Convert a list of parts and shapely objects to a positive resist design by adding a buffer around the actual design.

Parameters

- **parts** – List of parts and shapely objects.
- **buffer_radius** – Buffer radius
- **outer_resolution** – Outer buffer circumference resolution. Defaults to one 20th of the buffer radius.

- **clearance_features** – List of additional features to include in the generated structure. Can be useful for providing clearance areas around couplers or other features.
- **exclude** – List of features to subtract from the generated structure. Can be used for interconnects between structures from different cells, such that the end of a waveguide remains "open".

Returns Converted Shapely geometry.

Return type shapely.base.BaseGeometry

gdshelpers.helpers.small module

gdshelpers.helpers.small.**alphanumeric_to_id**(*text*)
Do the reverse of *id_to_alphanumeric*. :return: (column, row) tuple.

gdshelpers.helpers.small.**find_line_intersection**(*r1*, *angle1*, *r2*, *angle2*)
Find intersection between two lines defined by point and direction.

Parameters

- **r1** – Origin of the first line.
- **angle1** – Angle of the first line.
- **r2** – Origin of the second line.
- **angle2** – Angle of the second line.

Returns Tuple of point of intersection and distances from the origins.

Return type tuple

gdshelpers.helpers.small.**id_to_alphanumeric**(*column*, *row*)
Convert a *column*, *row* pair to an alphanumeric representation.

Parameters

- **column** (*int*) – Column
- **row** (*int*) – Row

Returns Alphanumeric representation.

Return type str

gdshelpers.helpers.small.**int_to_alphabet**(*num*)
Convert an integer number to an alphabetic representation.

Numbers of 0 to 25 are mapped to `A-Z`, higher numbers 'count' like `AA`, `AB`, . . . , `AZ`, `BA`. There is no upper limit on the converted number.

Parameters **num** – Number to convert

Returns Converted string.

Return type str

gdshelpers.helpers.small.**normalize_phase**(*phase*, *zero_to_two_pi=False*)
Normalize a phase to be within +/- pi.

Parameters

- **phase** (*float*) – Phase to normalize.
- **zero_to_two_pi** (*bool*) – True -> 0 to 2*pi, False -> +/- pi

Returns Normalized phase within +/- pi or 0 to 2*pi

Return type float

gdshelpers.helpers.small.**raith_eline_dosefactor_to_datatype**(*dose_factor*)
Convert a dose factor to a GDS datatype for Raith E-Beam writer.

Parameters **dose_factor** (*float*) – The dose factor.

Returns GDS datatype number

Return type int

gdshelpers.helpers.under_etching module

gdshelpers.helpers.under_etching.**create_holes_for_under_etching**(*underetch_parts*, *complete_structure*, *hole_radius*, *hole_distance*, *hole_spacing*, *hole_length=0*, *cap_style='round'*)
Creates holes around given parts which can be used for underetching processes

Parameters

- **underetch_parts** – List of gdshelpers parts around which the holes shall be placed
- **complete_structure** – geometric union of the complete structure, needed to avoid collisions between underetching holes and other structures, e.g. waveguides
- **hole_radius** – Radius of the holes in microns
- **hole_distance** – Distance between the holes edges from the the structures in microns
- **hole_spacing** – Distance between the holes in microns
- **hole_length** – Length of the holes (if 0 creates circles, else rectangle like)
- **cap_style** – CAP_STYLE of the holes (i.e. 'round' or 'square', see Shapely Docs)

Returns Geometric union of the created holes

gdshelpers.helpers.vortex_traps module

gdshelpers.helpers.vortex_traps.**fill_waveguide_with_holes_in_honeycomb_lattice**(*waveguide*, *spacing*, *padding*, *hole_radius*)
Fills a given waveguide with holes which are arranged in a honeycomb structure This can be used for generating vortex traps like presented in https://doi.org/10.1103/PhysRevApplied.11.064053

Parameters

- **waveguide** – Waveguide to be filled
- **spacing** – Spacing between the holes
- **padding** – Minimum distance from the edge of the waveguide to the holes

- **hole_radius** – Radius of the holes

Returns Shapely object, which describes the holes

gdshelpers.helpers.vortex_traps.**surround_with_holes**(*geometry*, *hole_spacing*, *hole_radius*, *padding*, *max_distance*)

Surrounds the given geometry with holes, which are arranged in a square lattice around the structure. This can be used for generating vortex traps like presented in https://doi.org/10.1103/PhysRevApplied.11.064053

Parameters

- **geometry** – The geometry around which the holes are generated
- **hole_spacing** – Spacing between the holes
- **hole_radius** – Radius of the holes
- **padding** – Padding around the geometry
- **max_distance** – Maximum distance of a hole from the geometry

Returns Shapely object, which describes the holes

Module contents

gdshelpers.helpers.**int_to_alphabet**(*num*)

Convert an integer number to an alphabetic representation.

Numbers of 0 to 25 are mapped to `A`-`Z`, higher numbers 'count' like `AA`, `AB`, . . . , `AZ`, `BA`. There is no upper limit on the converted number.

Parameters **num** – Number to convert

Returns Converted string.

Return type str

gdshelpers.helpers.**id_to_alphanumeric**(*column*, *row*)

Convert a *column*, *row* pair to an alphanumeric representation.

Parameters

- **column** (*int*) – Column
- **row** (*int*) – Row

Returns Alphanumeric representation.

Return type str

gdshelpers.helpers.**normalize_phase**(*phase*, *zero_to_two_pi=False*)

Normalize a phase to be within +/- pi.

Parameters

- **phase** (*float*) – Phase to normalize.
- **zero_to_two_pi** (*bool*) – True -> 0 to 2*pi, False -> +/- pi

Returns Normalized phase within +/- pi or 0 to 2*pi

Return type float

gdshelpers.helpers.**find_line_intersection**(*r1*, *angle1*, *r2*, *angle2*)

Find intersection between two lines defined by point and direction.

Parameters

- **r1** – Origin of the first line.
- **angle1** – Angle of the first line.
- **r2** – Origin of the second line.
- **angle2** – Angle of the second line.

Returns Tuple of point of intersection and distances from the origins.

Return type tuple

gdshelpers.helpers.**raith_eline_dosefactor_to_datatype**(*dose_factor*)

Convert a dose factor to a GDS datatype for Raith E-Beam writer.

Parameters **dose_factor** (*float*) – The dose factor.

Returns GDS datatype number

Return type int

gdshelpers.layout package

Submodules

gdshelpers.layout.grid module

class gdshelpers.layout.grid.**GridLayout**(*title=None*, *tight=False*, *region_layer_type='layout'*, *region_layer_on_labels=False*, *vertical_spacing=10*, *vertical_alignment=100*, *horizontal_spacing=10*, *horizontal_alignment=100*, *text_size=40*, *row_text_size=30*, *line_width=1*, *align_title_line=True*, *frame_layer=6*, *text_layer=7*, *region_layer=5*)

Bases: object

A grid layout class.

This class arranges cells in an ordered matter. It is useful if you want to place several independent devices next to each other.

Parameters

- **title** (*str*) – Title to be put on the top of the grid layout.
- **tight** (*bool*) – Pack all devices as close as possible along the x-axis.
- **region_layer_type** (*None, str*) – One of None, 'layout' or cell.
- **region_layer_on_labels** (*bool*) – Put a region layer around the labels as well when in cell mode.
- **vertical_spacing** (*float*) – Minimum vertical spacing between devices. The true spacing will be bigger in most cases, since it will align to the next write field.
- **vertical_alignment** (*float*) – Vertical alignment, typically the write field size.
- **horizontal_spacing** (*float*) – Minimum horizontal spacing between devices. The true spacing will be bigger in most cases, since it will align to the next write field.

- **horizontal_alignment** (*float*) – Horizontal alignment, typically the write field size.
- **text_size** (*float*) – Size of the title text.
- **row_text_size** (*float*) – Size of text placed in the rows.
- **line_width** (*float*) – Width of the frame.
- **frame_layer** (*int*) – Layer of the frame. If set to zero, the frame will not be drawn.
- **text_layer** (*int*) – Layer of text
- **region_layer** (*int*) – Layer of the region layer boxes.

add_column_label_row(*labels*, *row_label=None*, *size=None*, *alignment='center-top'*)
Start a new row, containing only labels.

Parameters

- **labels** (*tuple, list*) – List of labels for each column.
- **row_label** – Row label of the new row.
- **size** (*float*) – Size of the text.
- **alignment** – Alignment of the labels.

Type str, None

add_label_to_row(*text*, *size=None*, *origin=None*, *alignment='left-center'*)
Add a label to the current row.

Parameters

- **text** (*str*) – Text of the label.
- **size** (*float*) – Size of the label. Defaults to *row_text_size*.
- **origin** – Origin of the text.
- **alignment** – Alignment.

add_to_row(*cell=None*, *bbox=None*, *alignment='left-bottom'*, *realign=True*, *unique_id=None*, *allow_region_layer=True*)
Add a new cell to the row.

The dimensions of the cell will either be determined directly or can be given via *bbox*. They determined bounding box always assumes a start at (0, 0) to preserve write field alignment.

If *realign* is activated, the cell will be shifted, so that the it starts in the first write field quadrant. So if your cell contains structures as (-70, 10) and the write field size is 100, the cell will be shifted to (30, 10).

The alignment of the cell can also be changed, but note that only 'left-bottom' guarantees write field alignment. When the region layers are used, these will force an alignment again, though.

The position of each added cell inside the grid can be traced when passing a *unique_id* while adding the cell.

Parameters

- **cell** – Cell to add.
- **bbox** – Bounding box. If None, it will be determined based on *cell*.
- **alignment** – Alignment of the cell inside the grid.
- **realign** – Realign the cell, so that it starts in the first positive write field.

- **unique_id** – ID to trace where the cell was placed in the final layout.
- **allow_region_layer** (*bool*) – Allow a region layer around the cell.

Type gdsCAD.core.Cell

begin_new_row(*row_label=None*)
Begin a new row.

Parameters **row_label** (*str*) – Label of the row.

generate_layout(*cell_name='GRID_LAYOUT'*)
Generate a layout cell.

Parameters **cell_name** – Name of the generated layout cell

Returns Tuple of a cell, containing the layout and a dictionary mapping each unique id to the position inside the cell.

region_layer_type

gdshelpers.layout.write_field module

gdshelpers.layout.write_field.**annotate_write_fields**(*cell*, *origin='left-top'*, *end='right-bottom'*, *size=100*, *layer=4*)
Add write field marker lines to a cell.

Write fields are rectangular and typically start at the upper left part of a structure. This function adds write field lines to a given cell. The bounds are either found automatically or can be given as (x, y) tuple.

Alignment options are given as - separated tuple, allowing for combinations of `left`, `right` with `bottom`, `top`.

Parameters

- **cell** – The cell in which the lines will be added.
- **origin** – Origin of the write field markers.
- **end** – End of the write field markes. Must not be aligned to the write field, however.
- **size** – Size of the write field squares.
- **layer** – Layer on which the write field lines are added.

Module contents

class gdshelpers.layout.**GridLayout**(*title=None*, *tight=False*, *region_layer_type='layout'*, *region_layer_on_labels=False*, *vertical_spacing=10*, *vertical_alignment=100*, *horizontal_spacing=10*, *horizontal_alignment=100*, *text_size=40*, *row_text_size=30*, *line_width=1*, *align_title_line=True*, *frame_layer=6*, *text_layer=7*, *region_layer=5*)
Bases: `object`

A grid layout class.

This class arranges cells in an ordered matter. It is useful if you want to place several independent devices next to each other.

Parameters

- **title** (*str*) – Title to be put on the top of the grid layout.
- **tight** (*bool*) – Pack all devices as close as possible along the x-axis.
- **region_layer_type** (*None, str*) – One of `None`, `'layout'` or `cell`.
- **region_layer_on_labels** (*bool*) – Put a region layer around the labels as well when in `cell` mode.
- **vertical_spacing** (*float*) – Minimum vertical spacing between devices. The true spacing will be bigger in most cases, since it will align to the next write field.
- **vertical_alignment** (*float*) – Vertical alignment, typically the write field size.
- **horizontal_spacing** (*float*) – Minimum horizontal spacing between devices. The true spacing will be bigger in most cases, since it will align to the next write field.
- **horizontal_alignment** (*float*) – Horizontal alignment, typically the write field size.
- **text_size** (*float*) – Size of the title text.
- **row_text_size** (*float*) – Size of text placed in the rows.
- **line_width** (*float*) – Width of the frame.
- **frame_layer** (*int*) – Layer of the frame. If set to zero, the frame will not be drawn.
- **text_layer** (*int*) – Layer of text
- **region_layer** (*int*) – Layer of the region layer boxes.

add_column_label_row(*labels*, *row_label=None*, *size=None*, *alignment='center-top'*)
Start a new row, containing only labels.

Parameters

- **labels** (*tuple, list*) – List of labels for each column.
- **row_label** – Row label of the new row.
- **size** (*float*) – Size of the text.
- **alignment** – Alignment of the labels.

Type str, None

add_label_to_row(*text*, *size=None*, *origin=None*, *alignment='left-center'*)
Add a label to the current row.

Parameters

- **text** (*str*) – Text of the label.
- **size** (*float*) – Size of the label. Defaults to *row_text_size*.
- **origin** – Origin of the text.
- **alignment** – Alignment.

add_to_row(*cell=None*, *bbox=None*, *alignment='left-bottom'*, *realign=True*, *unique_id=None*, *allow_region_layer=True*)
Add a new cell to the row.

The dimensions of the cell will either be determined directly or can be given via *bbox*. They determined bounding box always assumes a start at (0, 0) to preserve write field alignment.

If *realign* is activated, the cell will be shifted, so that the it starts in the first write field quadrant. So if your cell contains structures as (-70, 10) and the write field size is 100, the cell will be shifted to (30, 10).

The alignment of the cell can also be changed, but note that only 'left-bottom' guarantees write field alignment. When the region layers are used, these will force an alignment again, though.

The position of each added cell inside the grid can be traced when passing a *unique_id* while adding the cell.

Parameters

- **cell** – Cell to add.
- **bbox** – Bounding box. If None, it will be determined based on *cell*.
- **alignment** – Alignment of the cell inside the grid.
- **realign** – Realign the cell, so that it starts in the first positive write field.
- **unique_id** – ID to trace where the cell was placed in the final layout.
- **allow_region_layer** (*bool*) – Allow a region layer around the cell.

Type gdsCAD.core.Cell

begin_new_row(*row_label=None*)
Begin a new row.

Parameters **row_label** (*str*) – Label of the row.

generate_layout(*cell_name='GRID_LAYOUT'*)
Generate a layout cell.

Parameters **cell_name** – Name of the generated layout cell

Returns Tuple of a cell, containing the layout and a dictionary mapping each unique id to the position inside the cell.

region_layer_type

gdshelpers.layout.**annotate_write_fields**(*cell*, *origin='left-top'*, *end='right-bottom'*, *size=100*, *layer=4*)
Add write field marker lines to a cell.

Write fields are rectangular and typically start at the upper left part of a structure. This function adds write field lines to a given cell. The bounds are either found automatically or can be given as (x, y) tuple.

Alignment options are given as - separated tuple, allowing for combinations of `left`, `right` with `bottom`, `top`.

Parameters

- **cell** – The cell in which the lines will be added.
- **origin** – Origin of the write field markers.
- **end** – End of the write field markes. Must not be aligned to the write field, however.
- **size** – Size of the write field squares.
- **layer** – Layer on which the write field lines are added.

gdshelpers.parts package

Submodules

gdshelpers.parts.cavity module

class gdshelpers.parts.cavity.**PhotonicCrystalCavity**(*origin*, *angle*, *width*, *lengthofcavity*, *numberofholes=14*, *holediameters=0.5*, *holedistances=0.5*, *tapermode=None*, *finalwidth=None*, *taperlength=None*, *samplepoints=1000*, *holeparams=None*, *underetching=True*, *markers=True*)

Bases: `object`

class to implement standard Photonic Crystal Cavity devices with GDS helpers see below for examples how to implement cavities with 1. constant hole diameters/distances 2. varying hole diameters/distances 3. tapered waveguide width (set cavity length to - holediameter)

generate_marker()

generate_underetch()

get_holes_list()

get_left_port()

get_right_port()

classmethod make_at_port(*port*, ***kwargs*)

gdshelpers.parts.coupler module

class gdshelpers.parts.coupler.**GratingCoupler**(*origin*, *angle*, *width*, *full_opening_angle*, *grating_lines*, *n_points=300*, *extra_triangle_layer=False*, *start_radius_absolute=False*)

Bases: `object`

A standard style radial grating coupler.

This coupler starts with the `width` of the input/output waveguide. The input is widened to the opening angle and ended with a circle which radius is given by the first value in the list of grating lines. The taper radius is either measured absolutely starting at the part origin or relative to the minimum radius. The minimum radius is the radius needed to achieve the opening angle.

All grating lines are given in a list which is the distance of the grating line to the last grating line. A list like `(10, 2, 5, 3, 6)` generates a coupler:

- with taper radius 10
- followed by a gap of 2 um and 5 um of material
- followed by a gap of 4 um and 5 um of material

In most cases it is too much work to calculate these grating intervals. The class functions *make_traditional_coupler()*, *make_traditional_coupler_at_port()* and *make_traditional_coupler_from_database()* will help you to create traditional couplers efficiently.

Parameters

- **origin** (*list, tuple*) – 2-element list or tuple specifying the `(x, y)` coordinates of the coupler.

- **width** (*float*) – With of the waveguide
- **full_opening_angle** (*float*) – Full opening angle of the coupler in radiant.
- **grating_lines** – List specifying the taper radius and the position of the grating edges.
- **n_points** – Number of points used per grating edge.
- **extra_triangle_layer** (*bool*) – If True, the taper triangle is painted separately and can be accessed via *get_shapely_object_triangle()*. Disabled by default.
- **angle** – Angle of the coupler in radiant. Defaults to -pi/2, hence the waveguide of the coupler points upwards.
- **start_radius_absolute** – If set to True, the first element of `grating_lines` is the absolute radius of the taper measured to its origin. An assertion error is thrown, if the radius is to small to achieve the full opening angle.

get_description_str(*grating_period=True*, *fill_factor=True*, *taper_length=True*, *opening_angle=True*)

get_description_text(*height=3.0*, *space=10*, *side='right'*, ***desc_options*)

get_shapely_object()
Get the Shapely object of this coupler.

If requested during creation of the coupler, the tapered triangle is not included in this Shapely object. It has to be acquired via *get_shapely_object_triangle()*.

Returns A Shapely object.

get_shapely_object_triangle()
Get the Shapely object of this coupler, but only the tapered triangle.

Returns A Shapely object.

classmethod make_traditional_coupler(*origin*, *width*, *full_opening_angle*, *grating_period*, *grating_ff*, *n_gratings*, *ap_max_ff=1.0*, *n_ap_gratings=0*, *taper_length=None*, *extra_triangle_layer=False*, *angle=-1.5707963267948966*, *n_points=394*, *implement_cadence_ff_bug=True*, *ap_start_period=None*)
Generate a traditional coupler as the `lib/coupler/` functions did. But this version is versatile enough to generate all kinds of couplers, including:

- constant fill factor couplers
- **apodized couplers (where the fill factor is varied from ap_max_ff to grating_ff and the grating period** is varied from ap_start_period to grating_period over n_ap_gratings gratings.)

All couplers can have the taper triangle separated onto its own layer for better e-beam dose.

Parameters

- **origin** (*list, tuple*) – 2-element list or tuple specifying the `(x, y)` coordinates of the coupler.
- **width** (*float*) – With of the waveguide
- **full_opening_angle** (*float*) – Full opening angle of the coupler in radiant.
- **grating_period** (*float*) – The grating period of the coupler.
- **grating_ff** (*float*) – The fill factor of the coupler between 0 and 1.

- **n_gratings** (*int*) – Number of gratings.
- **ap_max_ff** – Maximum fill factor of apodized gratings.
- **n_ap_gratings** (*int*) – Number of apodized gratings.
- **taper_length** – If not None, this sets the radius if the inner taper. Otherwise the length will be equal to the grating length.
- **n_points** – Number of points used per grating edge.
- **extra_triangle_layer** (*bool*) – If True, the taper triangle is painted seperatly and can be accessed via *get_shapely_object_triangle()*. Disabled by default.
- **angle** – Angle of the coupler in radiant. Defaults to -pi/2, hence the waveguide of the coupler points upwards.
- **implement_cadence_ff_bug** (*bool*) –
- **ap_start_period** – The starting period in the apodized region. If a value is passed, the period will be swept from *ap_start_period* to *grating_period* over the apodized gratings. If None, the period will be constant.

Returns The constructed traditional grating coupler.

Return type *GratingCoupler*

classmethod make_traditional_coupler_at_port(*port*, ***kwargs*)

Make a traditional coupler at a port.

This function is identical to *make_traditional_coupler()*. Parameters of the port can also be overwritten via keyword arguments.

Parameters

- **port** (*Port*) – The port at which the coupler shall be created.
- **kwargs** – Keyword arguments passed to *make_traditional_coupler()*.

Returns The constructed traditional grating coupler.

Return type *GratingCoupler*

classmethod make_traditional_coupler_from_database(*origin*, *width*, *db_id*, *wavelength*, ***kwargs*)

classmethod make_traditional_coupler_from_database_at_port(*port*, *db_id*, *wavelength*, ***kwargs*)

maximal_radius

origin

port

The port of the coupler.

Returns The coupler port.

Return type *Port*

width

gdshelpers.parts.coupler_references module

This module includes a collection of coupler parameters. It is not meant to be used directly but rather via the `GratingCoupler.make_traditional_coupler_from_database()` functions.

Feel free to report your coupler findings for inclusion in this database.

gdshelpers.parts.image module

gdshelpers.parts.interferometer module

class gdshelpers.parts.interferometer.**MachZehnderInterferometer**(*origin*, *angle*, *width*, *splitter_length*, *splitter_separation*, *bend_radius*, *upper_vertical_length*, *lower_vertical_length*, *horizontal_length*)

Bases: `object`

A simple Mach-Zehnder interferometer based on Y-splitters.

Parameters

- **origin** (*tuple*) – Start of the interferometer.
- **angle** (*float*) – Angle of the interferometer in rad.
- **width** (*float*) – Waveguide width.
- **splitter_length** (*float*) – Length of the splitter
- **splitter_separation** (*float*) – Separation of the slitter branches.
- **bend_radius** (*float*) – Bend radius.
- **upper_vertical_length** (*float*) – Straight length of the upper branch.
- **lower_vertical_length** (*float*) – Straight length of the lower branch.
- **horizontal_length** (*float*) – Straight horizontal length for both branches.

device_width

get_shapely_object()

classmethod make_at_port(*port*, *splitter_length*, *splitter_separation*, *bend_radius*, *upper_vertical_length*, *lower_vertical_length*, *horizontal_length*)

port

class gdshelpers.parts.interferometer.**MachZehnderInterferometerMMI**(*origin*, *angle*, *width*, *splitter_length*, *splitter_width*, *bend_radius*, *upper_vertical_length*, *lower_vertical_length*, *horizontal_length*)

Bases: object

A simple Mach-Zehnder interferometer based on Y-splitters.

Parameters

- **origin** (*tuple*) – Start of the interferometer.
- **angle** (*float*) – Angle of the interferometer in rad.
- **width** (*float*) – Waveguide width.
- **splitter_length** (*float*) – Length of the splitter
- **splitter_separation** (*float*) – Separation of the slitter branches.
- **bend_radius** (*float*) – Bend radius.
- **upper_vertical_length** (*float*) – Straight length of the upper branch.
- **lower_vertical_length** (*float*) – Straight length of the lower branch.
- **horizontal_length** (*float*) – Straight horizontal length for both branches.

device_width

get_shapely_object()

classmethod make_at_port(*port*, *splitter_length*, *splitter_width*, *bend_radius*, *upper_vertical_length*, *lower_vertical_length*, *horizontal_length*)

port

gdshelpers.parts.logo module

class gdshelpers.parts.logo.**KITLogo**(*origin*, *height*, *min_radius_fraction=0.05*)

Bases: object

A simplified logo of the Karlsruhe Institute of Technology (KIT).

Parameters

- **origin** – Tuple specifying the lower left corner of the logo.
- **height** – Height of the logo.
- **min_radius_fraction** – To avoid steep angles, the rays are cut at the minimum radius fraction.

get_shapely_object()

class gdshelpers.parts.logo.**WWULogo**(*origin*, *height*, *text*)
Bases: object

WWU Logo with WWU written next to it

Parameters

- **origin** – Tuple specifying the lower left corner of the logo.
- **height** – Height of the logo.
- **text** – 0 no text, 1 text right of logo, 2 text below logo

get_shapely_object()

gdshelpers.parts.marker module

class gdshelpers.parts.marker.**AutoStigmationMarker**(*origin*, *maximum_feature_size=3.0*, *minimum_feature_size=0.1*, *reduction_factor=1.2*, *resolution=16*)
Bases: object

get_shapely_object()

class gdshelpers.parts.marker.**CrossMarker**(*origin*, *cross_length*, *cross_width*, *paddle_length*, *paddle_width*)
Bases: object

Simple cross type marker with support for paddles.

This code is an adapted version of Nico's marker.

get_shapely_object()

classmethod make_simple_cross(*origin*, *cross_length*, *cross_width*)

classmethod make_traditional_paddle_markers(*origin*, *scale=1.0*, *cross_length_factor=2.0*, *cross_width_factor=0.1*, *paddle_length_factor=3.0*, *paddle_width_factor=0.5*)

class gdshelpers.parts.marker.**DLWMarker**(*origin*, *box_size=2.5*)
Bases: object

get_shapely_object()

class gdshelpers.parts.marker.**DLWPrecisionMarker**(*origin*, *size*, *frame_width*)
Bases: object

A specific marker to test the writing accuracy/alignment of the Nanoscribe DLW machine.

Parameters

- **origin** – Position of the marker.
- **size** – Size of the marker.
- **frame_width** – Size of the marker frame.

get_shapely_object()

class gdshelpers.parts.marker.**SquareMarker**(*origin*, *size*)
Bases: object

get_shapely_object()

classmethod make_marker(*origin*, *size=20*)

gdshelpers.parts.mode_converter module

class gdshelpers.parts.mode_converter.**StripToSlotModeConverter**(*origin*, *angle*, *width*, *taper_length*, *final_width*, *pre_taper_length*, *pre_taper_width*)

Bases: object

Generates a strip to slot mode converter as presented by Palmer et. al https://doi.org/10.1109/JPHOT.2013.2239283. If the input port width is a scalar und the final width is an array, a strip to slot mode converter is generated. On the other hand, if the input port width is an array and the final width is a scaler, a slot to strip mode converter is generated.

get_shapely_object()

Generates the mode converter. If the input port width is a scalar und the final width is an array, a strip to slot mode converter is generated. On the other hand, if the input port width is an array and the final width is a scaler, a slot to strip mode converter is generated.

Returns shapely object

in_port

Returns the input port of the mode converter

Returns port

classmethod make_at_port(*port*, *taper_length*, *final_width*, *pre_taper_length*, *pre_taper_width*)

Parameters

- **port** – port of the taper (origin, angle, width)
- **taper_length** – length of the taper
- **final_width** – final width of the mode converter. Array if strip to slot, scalar if slot to strip
- **pre_taper_length** – length of the pre taper
- **pre_taper_width** – width of the pre taper

Returns

out_port

Returns the output port of the mode converter

Returns port

gdshelpers.parts.ntron module

class gdshelpers.parts.ntron.**Ntron**(*origin*, *angle*, *gate_width_1*, *gate_width_2*, *choke_width_1*, *choke_width_2*, *choke_length_2*, *choke_length_3*, *choke_point_2=(0.5, 0)*, *choke_point_3=(1, 0.5)*, *points_per_curve=1000*, *gate_start=0*, *gate_point_2=(0.3333333333333333, 0)*, *gate_point_3=(0.6666666666666666, 1)*, *channel_point_2=(0.3333333333333333, 0)*, *channel_point_3=(0.6666666666666666, 0)*, *channel_length=2.5*, *outer_channel_width=0.4*, *inner_channel_width=0.2*, *channel_position=0.5*, *gate_length=0.5*, *choke_start=0.5*, *gate_choke_length=0.2*)

Bases: `object`

angle

get_shapely_object()

classmethod make_at_port_(*port*, *gate_width_1*, *gate_width_2*, *choke_width_1*, *choke_width_2*, *choke_length_2*, *choke_length_3*, *choke_point_2=(0.5, 0)*, *choke_point_3=(1, 0.5)*, *points_per_curve=1000*, *gate_start=0*, *gate_point_2=(0.3333333333333333, 0)*, *gate_point_3=(0.6666666666666666, 1)*, *channel_point_2=(0.3333333333333333, 0)*, *channel_point_3=(0.6666666666666666, 0)*, *channel_length=2.5*, *outer_channel_width=0.4*, *inner_channel_width=0.2*, *channel_position=0.3*, *gate_length=0.5*, *choke_start=0.5*, *gate_choke_length=0.2*, *target='_gate'*)

Class method to directly place a ntron to a given port, connected to a chosen port

Parameters

- **port** – port from the connecting structure
- **gate_width_1** – outer width of the _gate port
- **gate_width_2** – connection gate-choke, should be same as choke_width_1
- **choke_width_1** – outer width of the choke
- **choke_width_2** – Inner choke width, which is the bottleneck and the most important parameter
- **choke_length_2** – length of the gate sided choke bezier shape
- **choke_length_3** – length of the channel sided choke bezier shape
- **choke_point_2** – 2nd bezier point in a rectangle (Look Bezier explanation above)
- **choke_point_3** – 3rd bezier point in a rectangle (Look Bezier explanation above)
- **points_per_curve** – number of of points per Bezier curve used to create a shapely geometry
- **gate_start** – starting x pos of the gate section, should be zero
- **gate_point_2** – 2nd bezier point in a rectangle (Look Bezier explanation above)
- **gate_point_3** – 3rd bezier point in a rectangle (Look Bezier explanation above)
- **channel_point_2** – 2nd bezier point in a rectangle (Look Bezier explanation above)

- **channel_point_3** – 3rd bezier point in a rectangle (Look Bezier explanation above)
- **channel_length** – length of the channel
- **outer_channel_width** – starting width of the channel at the gates
- **inner_channel_width** – end width in the middle of the channel
- **channel_position** – channel position corresponding to the gate, for 0 choke is on the bottom, for 1 on the top
- **gate_length** – the length of the bezier calculated shape of the gate
- **choke_start** – starting x pos of the choke section, should be equal to gate_length
- **gate_choke_length** –
- **target** – the ntron port to connect to

Returns

origin

port_drain

port_gate

port_source

width

gdshelpers.parts.ofwa module

Parts for Optical Field Writable Arrays (OFWA)

class gdshelpers.parts.ofwa.**MultiPortSwitch**(*origin*, *angle*, *in_ports*, *out_ports*, *port_spacing*, *taper_length*, *taper_function*, *radius*, *wg_bend_radius*, *displacement=0.0*, *minimal_final_spacing=None*)

Bases: object

angle

dlw_in_ports

dlw_out_ports

get_dlw_in_port(*idx*)

get_dlw_out_port(*idx*)

get_in_port(*idx*)

get_out_port(*idx*)

get_shapely_object()

in_ports

classmethod make_at_in_port(*port*, *in_port_idx*, ***kwargs*)

classmethod make_at_out_port(*port*, *out_port_idx*, ***kwargs*)

marker_positions

origin

out_ports

gdshelpers.parts.optical_codes module

class gdshelpers.parts.optical_codes.**QRCode**(*origin*, *data*, *box_size*, *version=None*, *error_correction=0*, *border=0*, *alignment='left-bottom'*)

Bases: object

Quick Response (QR) code part.

This part is a simple wrapper around the qrcode library using the ShapelyImageFactory. If you need more flexibility this class might be extended or you can use the shapely output factory ShapelyImageFactory and qrcode directly.

Parameters

- **origin** – Lower left corner of the QR code.
- **data** – Data which is to be encoded into the QR code.
- **box_size** – Size of each box.
- **version** – QR code version.
- **error_correction** – Level of error correction.
- **border** – Size of the surrounding free border.

ERROR_CORRECT_H = 2

ERROR_CORRECT_L = 1

ERROR_CORRECT_M = 0

ERROR_CORRECT_Q = 3

get_shapely_object()

class gdshelpers.parts.optical_codes.**ShapelyImageFactory**(*border*, *width*, *box_size*, **args*, ***kwargs*)

Bases: qrcode.image.base.BaseImage

Output factory for qrcode, generating Shapely polygons.

Probably you will not need to use it directly. The *QRCode* class provides a simple to use part based on this image factory.

drawrect(*row*, *col*)
Draw a single rectangle of the QR code.

get_shapely_object()

kind = 'shapely'

new_image(*origin=(0, 0)*, *scale_factor=0.001*, ***kwargs*)
Build the image class. Subclasses should return the class created.

origin

size

gdshelpers.parts.pattern_import module

gdshelpers.parts.port module

class gdshelpers.parts.port.**Port**(*origin*, *angle*, *width*)
Bases: object

Abstraction of a waveguide port.

Other objects might dock to a port. It is simply a helper object to allow easy chaining of parts.

Parameters

- **origin** – Origin of the port.
- **angle** – Angle of the port.
- **width** (*float*) – Width of the port.

angle
The angle of the port.

copy()
Create a copy if the port.

Returns A copy of the port.

Return type *Port*

debug_shape

get_parameters()
Get a dictionary representation of the port properties.

Returns A dictionary containing the origin, angle and width of the port.

Return type dict

inverted_direction
Get a port which points in the opposite direction.

Returns A copy of this port, pointing in the opposite direction.

Return type *Port*

longitudinal_offset(*offset*)
Returns a new port, which offset in in direction of this port.

Parameters **offset** (*float*) – Offset from the end of the port. Positive is the direction, the port is pointing.

Returns The new offset port

Return type *Port*

origin
The origin coordinates of this port.

When reading it is guarantied to be a 2-dim numpy array.

parallel_offset(*offset*)
Returns a new port, which offset in parallel from this port.

Parameters **offset** (*float*) – Offset from the center of the port. Positive is left of the port.

Returns The new offset port

Return type *Port*

rotated(*angle*)
Returns a new port, which is rotated by the given angle.

Parameters **angle** (*float*) – Angle to rotate.

Returns The new rotated port

Return type *Port*

set_port_properties(***kwargs*)
Set port parameters via named keyword arguments.

Parameters **kwargs** – The keywords to set.

Returns The modified port

Return type *Port*

total_width
The total width of the port.

Guarantied to be a positive float.

width
The width of the port. E.g. for slot waveguides it can also be an array in the format [width, gap, width, ...], where each width describe the width of each rail and the gap defines the gap in between. This array can also end with a gap, which facilitates e.g. the design of adiabatic mode converters.

with_width(*width*)
Returns a new port, of which the width is set to the new value

Parameters **width** – Width of the resulting port

Returns The new port

Return type *Port*

x

y

gdshelpers.parts.resonator module

Ring and race track resonators

class gdshelpers.parts.resonator.**RingResonator**(*origin*, *angle*, *width*, *gap*, *radius*, *race_length=0*, *draw_opposite_side_wg=False*, *res_wg_width=None*, *n_points=None*, *straight_feeding=False*, *vertical_race_length=0*)

Bases: object

A simple Ring / Race track resonator.

This part implements a super simple ring resonator with optional race tracks. Several helper functions are available to calculate points and ports of interest. The width of the feeding waveguide and the ring waveguide may differ.

Parameters

- **origin** – Origin of the resonator, which is the start of the input waveguide.
- **angle** – Angle of the input waveguide.

- **width** – Width of the angle waveguide.
- **gap** – Gap between ring and waveguide. If positive, the ring will be on the left, and on the right side for negative gap values. Can also be a 2-tuple, if input and output gap should be different.
- **radius** – Radius of the bends.
- **race_length** – Length of the race track. Defaults to zero.
- **draw_opposite_side_wg** – If True, draw the opposing waveguides, (a.k.a. drop ports.)
- **res_wg_width** – Width of the resonator waveguide. If None, the width of the input waveguide is assumend.
- **n_points** – Number of points used per quarter circle. If None, it uses the bend default.
- **straight_feeding** – Add straight connections on both sides of the resonator.
- **vertical_race_length** – Length of a vertical race track section. Defaults to zero.

add_port

angle

center_coordinates

circumference

drop_port

get_shapely_object()

in_port

classmethod make_at_port(*port*, *gap*, *radius*, ***kwargs*)

opposite_side_port_in

opposite_side_port_out

origin

out_port

port

through_port

width

gdshelpers.parts.snspd module

class gdshelpers.parts.snspd.**SNSPD**(*origin*, *angle*, *width*, *nano_wire_width*, *nano_wire_gap*, *nano_wire_length*, *waveguide_tapering*, *passivation_buffer*)

Bases: object

current_port

get_passivation_layer()

get_shapely_object()

get_waveguide()

left_electrode_port

classmethod make_at_port(*port*, *nw_width*, *nw_gap*, *nw_length*, *waveguide_tapering*, *passivation_buffer*)

right_electrode_port

gdshelpers.parts.source module

class gdshelpers.parts.source.**CNT**(*origin*, *angle*, *width*, *gap=0.15*, *l_taper=10*, *w_taper=0.9*, *el_l_straight=8*, *el_l_taper=2.5*, *el_final_width=2*, *el_l_fine=1.4*, *el_radius=0.1*, *n_points=128*)

Bases: object

Creates a CNT source (Electrodes + tapered waveguide) at waveguide port.

This class implements the electrodes and tapered waveguide for an integrated CNT source. It provides the electrodes ports and the connecting waveguide port. The Electrode is made up of three part: the round head (defined by the radius) followed by a straight fine line of length el_l_fine, followed by a taper to final_width with length el_l_taper and a box with length el_l_straight.

Parameters

- **origin** – Origin of the resonator, which is the start of the input waveguide.
- **angle** – Angle of the input waveguide.
- **width** – Width of the angle waveguide.
- **gap** – Gap between electrode tip and waveguide.
- **l_taper** – length of the waveguide taper used before and after the cnt. i.a. 2*l_taper will be added to the waveguide
- **w_taper** – width of waveguide at the cnts position
- **el_l_straight** – length of electrode part with largest thickness
- **el_l_taper** – length over which 2*el_radius is tapered to el_final_width
- **el_final_width** – largest width of the electrode
- **el_l_fine** – length of small strip to the tip of the electrode
- **el_radius** – radius of electrodes tip
- **n_points** – number of points used to make electrode tip polygon

get_shapely_object()

in_port

left_electrode_port

classmethod make_at_port(*port*, ***kwargs*)

out_port

right_electrode_port

gdshelpers.parts.spiral module

class gdshelpers.parts.spiral.**Spiral**(*origin*, *angle*, *width*, *num*, *gap*, *inner_gap*)

Bases: object

angle

get_shapely_object()

in_port

length

classmethod make_at_port(*port*, *num*, *gap*, *inner_gap*)
Creates a Spiral around the given port

Parameters

- **port** – port at which the spiral starts
- **num** – number of turns
- **gap** – gap between two waveguides
- **inner_gap** – inner radius of the spiral

origin

out_port

width

gdshelpers.parts.splitter module

class gdshelpers.parts.splitter.**DirectionalCoupler**(*origin*, *angle*, *wg_width*, *length*, *gap*, *bend_radius*, *bend_angle=0.6283185307179586*)

Bases: object

get_shapely_object()

classmethod make_at_port(*port*, *length*, *gap*, *bend_radius*, *bend_angle=0.6283185307179586*, *which=0*)
Creates a dc coupler starting at the port :param port: starting port :param length: coupling length :param gap: gap between the waveguides in the coupling area :param bend_radius: radius of the curves :param bend_angle: angle of the curves :param which: decides on which side to start, either 0 or 1 :return:

class gdshelpers.parts.splitter.**MMI**(*origin*, *angle*, *wg_width*, *length*, *width*, *num_inputs*, *num_outputs*, *taper_width=2*, *taper_length=10*)

Bases: object

get_shapely_object()

left_branch_port
Returns the leftmost output port (like a Y-Splitter)

Returns leftmost Port

classmethod make_at_port(*port*, *length*, *width*, *num_inputs*, *num_outputs*, *pos='i0'*, *taper_width=2*, *taper_length=10*)
Creates a Multimode Interference Coupler at the given port :param port: port to make the coupler at :param length: length of the multimode area :param width: width of the multimode area :param num_inputs: number of input ports :param num_outputs: number of output ports :param pos: port, that has to be connected to the given port, first letter-> i=input/o=output, second letter->number of the input/output :param taper_width: width of the tapers at the multimode area :param taper_length: length of the tapers :return:

right_branch_port
Returns the rightmost output port (like a Y-Splitter)

Returns rightmost Port

separation

class gdshelpers.parts.splitter.**Splitter**(*origin*, *angle*, *total_length*, *wg_width_root*, *sep*, *wg_width_branches=None*, *n_points=50*, *implement_cadence_bug=False*)

Bases: object

get_shapely_object()

left_branch_port

classmethod make_at_left_branch_port(*port*, *total_length*, *sep*, *wavelength_root=None*, ***kwargs*)

classmethod make_at_right_branch_port(*port*, *total_length*, *sep*, *wavelength_root=None*, ***kwargs*)

classmethod make_at_root_port(*port*, *total_length*, *sep*, ***kwargs*)

right_branch_port

root_port

gdshelpers.parts.text module

class gdshelpers.parts.text.**Text**(*origin*, *height*, *text=''*, *alignment='left-bottom'*, *angle=0.0*, *font='stencil'*, *line_spacing=1.5*, *true_bbox_alignment=False*)

Bases: object

alignment

bounding_box

font

get_shapely_object()

height

origin

gdshelpers.parts.waveguide module

class gdshelpers.parts.waveguide.**Waveguide**(*origin*, *angle*, *width*)

Bases: object

add_arc(*final_angle*, *radius*, *final_width=None*, *n_points=128*, *shortest=True*, ***kwargs*)

add_bend(*angle*, *radius*, *final_width=None*, *n_points=128*, ***kwargs*)

add_bezier_to(*final_coordinates*, *final_angle*, *bend_strength*, *width=None*, ***kwargs*)

add_bezier_to_port(*port*, *bend_strength*, *width=None*, ***kwargs*)

add_cubic_bezier_path(*p0*, *p1*, *p2*, *p3*, *width=None*, ***kwargs*)

Add a cubic bezier path to the waveguide.

Coordinates are in the "waveguide tip coordinate system", so the first point will probably be p0 == (0, 0). Note that your bezier curve undergoes the same restrictions as a parameterized path. Don't self-intersect it and don't use small bend radii.

Parameters

- **p0** – 2 element tuple like coordinates
- **p1** – 2 element tuple like coordinates
- **p2** – 2 element tuple like coordinates
- **p3** – 2 element tuple like coordinates
- **width** – Width of the waveguide, as passed to :func:add_parameterized_path
- **kwargs** – Optional keyword arguments, passed to :func:add_parameterized_path

Returns Changed waveguide

Return type *Waveguide*

add_left_bend(*radius*, *angle=1.5707963267948966*)
Add a left turn (90° or as defined by angle) with the given bend radius

add_parameterized_path(*path*, *width=None*, *sample_distance=0.5*, *sample_points=100*, *path_derivative=None*, *path_function_supports_numpy=False*, *width_function_supports_numpy=False*)
Generate a parameterized path.

The path coordinate system is the origin and rotation of the current path. So if you want to continue your path start at (0, 0) in y-direction.

Note, that path is either a list of (x,y) coordinate tuples or a callable function which takes one float parameter between 0 and 1. If you use a parameterized function, its first derivative must be continuous. When using a list of coordinates, these points will be connected by straight lines. They must be sufficiently close together to simulate a first derivative continuous path.

This function will try to space the final points of the curve equidistantly. To achieve this, it will first sample the function and find its first derivative. Afterwards it can calculate the cumulative sum of the length of the first derivative. This allows to sample the function nearly equidistantly in a second step. This approach might be wasteful for paths like (x**2, y). You can suppress resampling for length by passing zero or none as sample_distance parameter.

The width of the generated waveguide may be constant when passing a number, or variable along the path when passing an array or a callable function, using the same parameter as the path. For generating slot/coplanar/... waveguides, start with a *Port* which has an array of the form *[rail_width_1, gap_width_1, rail_width_2, ...]* set as *width* and which defines the width of each rail and the gaps between the rails. This array is also allowed to end with a gap_width for positioning the rails asymmetrically to the path which can be useful e.g. for strip-to-slot mode converters.

Note, that your final direction of the path might not be what you expected. This is caused by the numerical procedure which generates numerical errors when calculating the first derivative. You can either append another arc to the waveguide to get to you a correct angle or you can also supply a function which is the algebraic first derivative. The returned vector is not required to be normed.

By default, for each parameter point t, the parameterized functions are call. You will notice that this is rather slow. To achieve the best performance, write your functions in such a way, that they can handle a numpy array as parameter *t*. Once the *path_function_supports_numpy* option is set to True, the function will be called only once, speeding up the calculation considerable.

Parameters

- **path** –
- **width** –
- **sample_distance** –

- **sample_points** –
- **path_derivative** –
- **path_function_supports_numpy** –
- **width_function_supports_numpy** –

add_right_bend(*radius*, *angle=1.5707963267948966*)
Add a right turn (90° or as defined by angle) with the given bend radius

add_route_single_circle_to(*final_coordinates*, *final_angle*, *final_width=None*, *max_bend_strength=None*, *on_line_only=False*)
Connect two points by straight lines and one circle.

Works for geometries like round edges and others. The final straight line can also be omitted so that the waveguide only end on the line described by the support vector and angle.

By default, this method tries to route to the target with the greatest possible circle. But the valid bending range may be limited via the *max_bend_strength* parameter.

This method does not work for geometries which cannot be connected only by straight lines and one circle, such as parallel lines etc.

Still, this method can prove extremely useful for routing to i.e. grating couplers etc.

Parameters

- **final_coordinates** – Final destination point.
- **final_angle** – Final angle of the waveguide.
- **final_width** – Final width of the waveguide.
- **max_bend_strength** – The maximum allowed bending radius.
- **on_line_only** – Omit the last straight line and only route to described line.

add_route_single_circle_to_port(*port*, *max_bend_strength=None*, *on_line_only=False*)
Connect to port by straight lines and one circle.

Helper function to conveniently call add_route_single_circle_to. :param port: Target port. :param max_bend_strength: The maximum allowed bending radius. :param on_line_only: Omit the last straight line and only route to line described by port.

add_route_straight_to_port(*port*)
Add a straight segment to a given port. The added segment will keep the angle of the current port at the start and use the angle of the target port at the end. If the ports are laterally shifted, this will result in a trapezoidal shape.

The width will be linearly tapered to that of the target port.

Parameters **port** – Target port.

add_straight_segment(*length*, *final_width=None*, ***kwargs*)

add_straight_segment_to_intersection(*line_origin*, *line_angle*, ***line_kw*)
Add a straight line until it intersects with an other line.

The other line is described by the support vector and the line angle.

Parameters

- **line_origin** – Intersection line support vector.
- **line_angle** – Intersection line angle.
- **line_kw** – Parameters passed on to add_straight_segment.

Raises **ArithmeticError** – When there is no intersection due to being parallel or if the intersection is behind the waveguide.

add_straight_segment_until_level_of_port(*port*, ***line_kw*)
Add a straight line until it is on the same level as the given port. If several ports are given in a list, the most distant port is chosen.

In this context "on the same level" means the intersection of the waveguide with the line orthogonal to the given port.

Parameters

- **port** – The port or a list of ports.
- **line_kw** –

add_straight_segment_until_x(*x*, ***line_kw*)
Add straight segment until the given x value is reached.

Parameters

- **x** – value
- **line_kw** – Parameters passed on to add_straight_segment.

add_straight_segment_until_y(*y*, ***line_kw*)
Add straight segment until the given y value is reached.

Parameters

- **y** – value
- **line_kw** – Parameters passed on to add_straight_segment.

angle

center_coordinates

current_port

get_segments()
Returns the list of tuples, containing their ports and shapely objects.

get_shapely_object()
Get a shapely object which forms this path.

get_shapely_outline()
Get a shapely object which forms the outline of the path.

in_port

length

length_last_segment

classmethod make_at_port(*port*, ***kargs*)

origin

port

width

x

y

Module contents

class gdshelpers.parts.**Port**(*origin*, *angle*, *width*)
Bases: object

Abstraction of a waveguide port.

Other objects might dock to a port. It is simply a helper object to allow easy chaining of parts.

Parameters

- **origin** – Origin of the port.
- **angle** – Angle of the port.
- **width** (*float*) – Width of the port.

angle
The angle of the port.

copy()
Create a copy if the port.

Returns A copy of the port.

Return type *Port*

debug_shape

get_parameters()
Get a dictionary representation of the port properties.

Returns A dictionary containing the origin, angle and width of the port.

Return type dict

inverted_direction
Get a port which points in the opposite direction.

Returns A copy of this port, pointing in the opposite direction.

Return type *Port*

longitudinal_offset(*offset*)
Returns a new port, which offset in in direction of this port.

Parameters **offset** (*float*) – Offset from the end of the port. Positive is the direction, the port is pointing.

Returns The new offset port

Return type *Port*

origin
The origin coordinates of this port.

When reading it is guarantied to be a 2-dim numpy array.

parallel_offset(*offset*)
Returns a new port, which offset in parallel from this port.

Parameters **offset** (*float*) – Offset from the center of the port. Positive is left of the port.

Returns The new offset port

Return type *Port*

rotated(*angle*)
Returns a new port, which is rotated by the given angle.

Parameters **angle** (*float*) – Angle to rotate.

Returns The new rotated port

Return type *Port*

set_port_properties(***kwargs*)
Set port parameters via named keyword arguments.

Parameters **kwargs** – The keywords to set.

Returns The modified port

Return type *Port*

total_width
The total width of the port.

Guarantied to be a positive float.

width
The width of the port. E.g. for slot waveguides it can also be an array in the format [width, gap, width, ...], where each width describe the width of each rail and the gap defines the gap in between. This array can also end with a gap, which facilitates e.g. the design of adiabatic mode converters.

with_width(*width*)
Returns a new port, of which the width is set to the new value

Parameters **width** – Width of the resulting port

Returns The new port

Return type *Port*

x

y

gdshelpers.simulation package

Submodules

gdshelpers.simulation.simulation module

Module contents

Module contents

3.6 Changelog

3.6.1 Unreleased

- get_reduced_layer() now considers cell-arrays

3.6.2 1.2.1

- Allow explicit definition of datatype by passing a tuple (layer, datatype) as layer
- Allow setting the width of LineStrings
- Angle of sub-cells considered in DLW-data and boundaries
- Fix GDSII-export: allow negative angles, correct rounding of spacing
- Allow specifying clearance features and exlusion features for convert_to_positive_resist

3.6.3 1.2.0

- Added meep integration
- GDSII-export: Added support for LineStrings

3.6.4 1.1.4

- Waveguide: added add_left_bend and add_right_bend to make code easier readable
- added alphanumeric_to_id as an inverse of id_to_alphanumeric
- allow to limit the numbers of workers for parallel export
- fixed oasis export

3.6.5 1.1.3

- Grating coupler: make_traditional_coupler now allows to apodize the period of the grating
- Port: added with_width function to generate a copy of the Port with a certain width
- Increased precision in add_straight_segment by evaluating derivative
- Added add_route_straight_to_port to Waveguide
- Fixed evaluation of width-parameter in add_parametrized_path
- Stopped testing with Python 3.5, as it reached it's end-of-life and added a warning
- Deprecated gdsCAD, as it isn't compatible with Python 3
- Fixed cell.show

3.6.6 1.1.2

- Added scale-parameter to save_image
- fixed .dxf-export in Cell
- Waveguide.add_parameterized_path now also supports an array as path_derivative
- fixed add_dlw_marker, origin can now also be a list

3.6.7 1.1.1

- Removed __future__ imports and (object) in class definitions for Python 2
- create_holes_for_under_etching now allows ovals and rectangles
- add_route_single_circle_to_port now tapers the waveguide to match the width of the port
- Bugfixes

3.6.8 1.1.0

- Added support for slot waveguides and coplanar waveguides
- Direct GDSII-export is now the standard GDSII-writer
- Added function for generating vortex traps
- Improved shape generation performance of waveguide
- Strip to slot mode converter added
- Bugfixes

3.6.9 1.0.4

- Added part for GDSII-import
- Added direct GDSII-export
- Added DXF-export
- bugfixes

3.6.10 1.0.3

- Structures in Cell are now converted individually for pattern export
- annotate_write_fields now works with Cells instead of gdscad.Cells
- fixed some bugs

3.6.11 1.0.2

- Dependencies are now installed automatically

3.6.12 1.0.1

- Project description now visible on PyPI

3.6.13 1.0.0

- Public release

Python Module Index

g

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

W

X

Y