Qpdk

gdsfactory

Sep 19, 2025

Contents

[1]

[2]

A generic process design kit (PDK) for superconducting quantum RF applications based on gdsfactory[7].

Examples

- PDK cells in the documentation[8]: showcases available geometries.
- *qpdk/samples/* (page **??**): contains example layouts and simulations.
- *qpdk/notebooks/* (page **??**): contains notebooks demonstrating design and simulation workflows.

Installation

We recommend using `uv`[9] for package management.

Installation for Users

Install the package with:

```
uv pip install qpdk
```

[!NOTE] After installation, restart KLayout to ensure the new technology appears.

Optional dependencies for the models and simulation tools can be installed with:

```
uv pip install qpdk[models]
```

Installation for Contributors

Clone the repository and install at least the development dependencies:

```
git clone https://github.com/gdsfactory/quantum-rf-pdk.git
cd quantum-rf-pdk
uv sync --extra dev
```

Testing and Building Documentation

Check out the commands for testing and building documentation with:

```
make help
```

[1] https://gdsfactory.github.io/quantum-rf-pdk/
[2] https://github.com/gdsfactory/quantum-rf-pdk/actions/workflows/test.yml
[7] https://gdsfactory.github.io/gdsfactory/
[8] https://gdsfactory.github.io/quantum-rf-pdk/cells.html
[9] https://astral.sh/uv/

Documentation

- Quantum RF PDK documentation (HTML)[10]
- Quantum RF PDK documentation (PDF)[11]
- gdsfactory documentation[12]

[10] https://gdsfactory.github.io/quantum-rf-pdk/
[11] https://gdsfactory.github.io/quantum-rf-pdk/qpdk.pdf
[12] https://gdsfactory.github.io/gdsfactory/

Part I

API

1 API

TODO

1.1 Cells QPDK

1.1.1 airbridge

qpdk.cells.**airbridge**(*bridge_length=30.0*, *bridge_width=8.0*, *pad_width=15.0*, *pad_length=12.0*, *bridge_layer=<LayerMapQPDK.AB_DRAW: 48>*, *pad_layer=<LayerMapQPDK.AB_VIA: 49>*)

Generate a superconducting airbridge component.

Creates an airbridge consisting of a suspended bridge span and landing pads on either side. The bridge allows transmission lines to cross over each other without electrical contact, which is essential for complex quantum circuit routing without crosstalk.

The bridge_layer (AB_DRAW) represents the elevated metal bridge structure, while the pad_layer (AB_VIA) represents the contact/landing pads that connect to the underlying circuit.

> **Note**
>
> To be used with `ComponentAlongPath` the unrotated version should be _oriented for placement on a horizontal **line_**.

Parameters

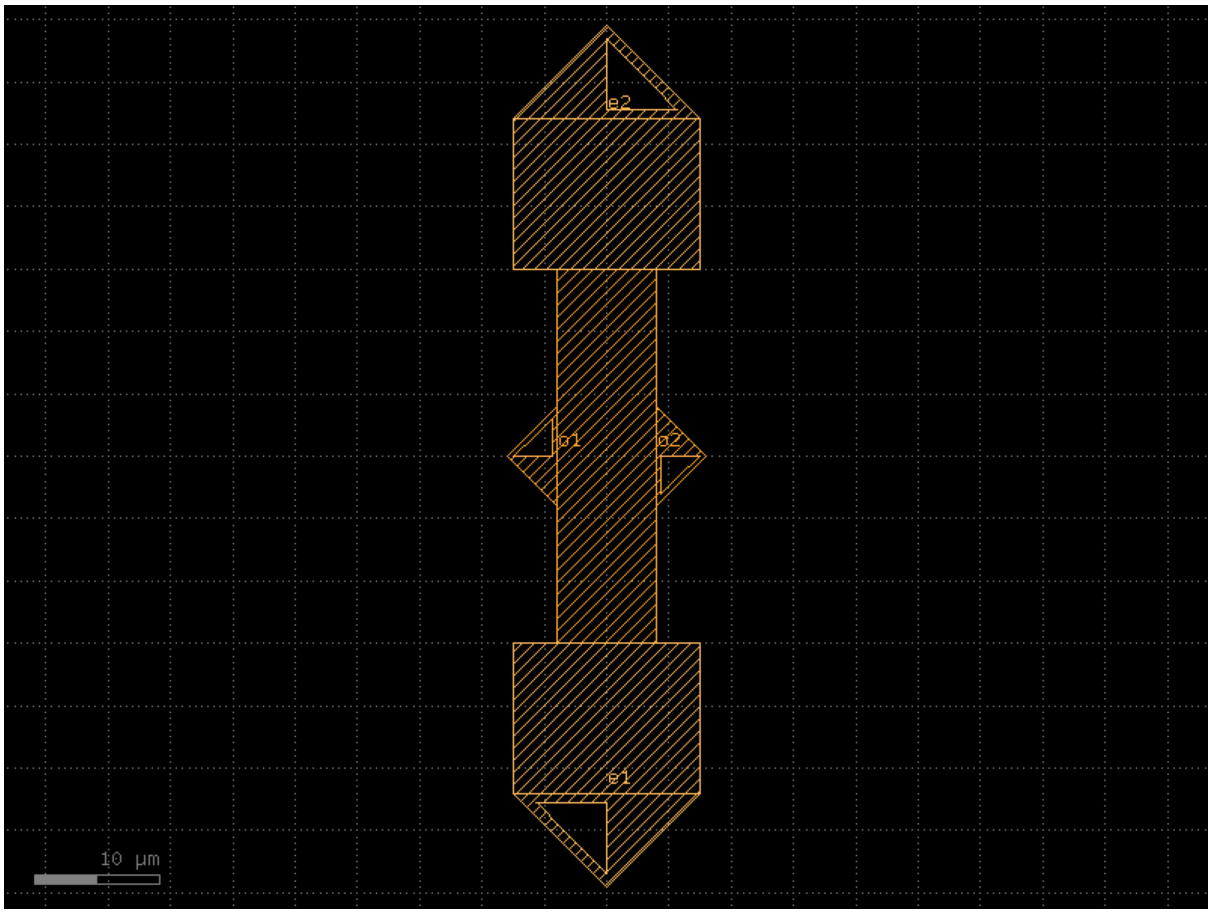
- **bridge_length** (*float*) – Total length of the airbridge span in µm.
- **bridge_width** (*float*) – Width of the suspended bridge in µm.
- **pad_width** (*float*) – Width of the landing pads in µm.
- **pad_length** (*float*) – Length of each landing pad in µm.
- **bridge_layer** (*LayerSpec*) – Layer for the suspended bridge metal (default: AB_DRAW).
- **pad_layer** (*LayerSpec*) – Layer for the landing pads/contacts (default: AB_VIA).

Returns
Component containing the airbridge geometry with appropriate ports.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.airbridge(bridge_length=30, bridge_width=8, pad_width=15, pad_length=12,␣
↪bridge_layer='AB_DRAW', pad_layer='AB_VIA').copy()
c.draw_ports()
c.plot()
```



1.1.2 bend_circular

qpdk.cells.**bend_circular**(***kwargs*)

Returns circular bend.

Parameters
****kwargs** (*Unpack[BendCircularKwargs]*) – Arguments passed to gf.c.bend_circular.

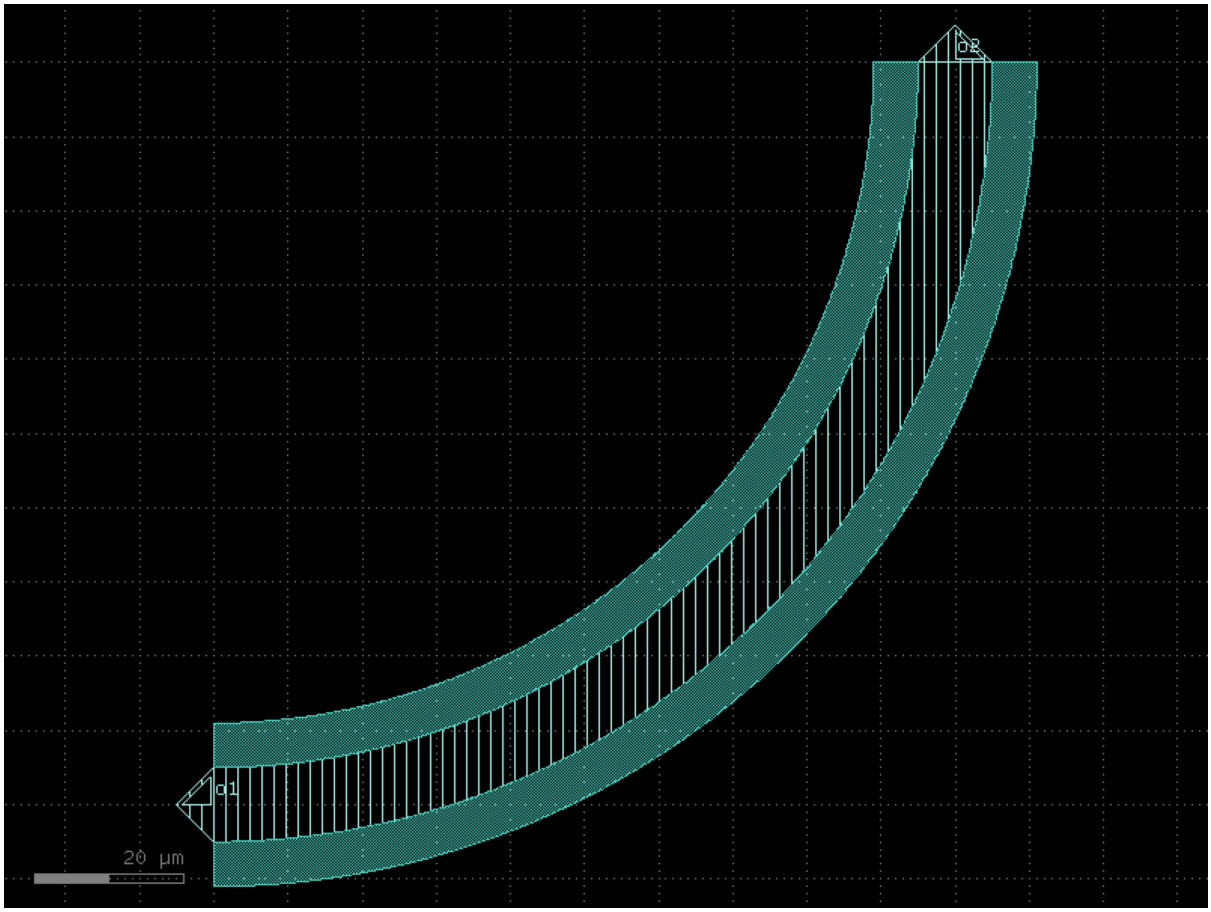
Return type
Component

```
from qpdk import cells, PDK
```

(continues on next page)

(continued from previous page)

```
PDK.activate()
c = cells.bend_circular().copy()
c.draw_ports()
c.plot()
```



1.1.3 bend_circular_all_angle

qpdk.cells.**bend_circular_all_angle**(***kwargs*)

Returns circular bend with arbitrary angle.

Parameters

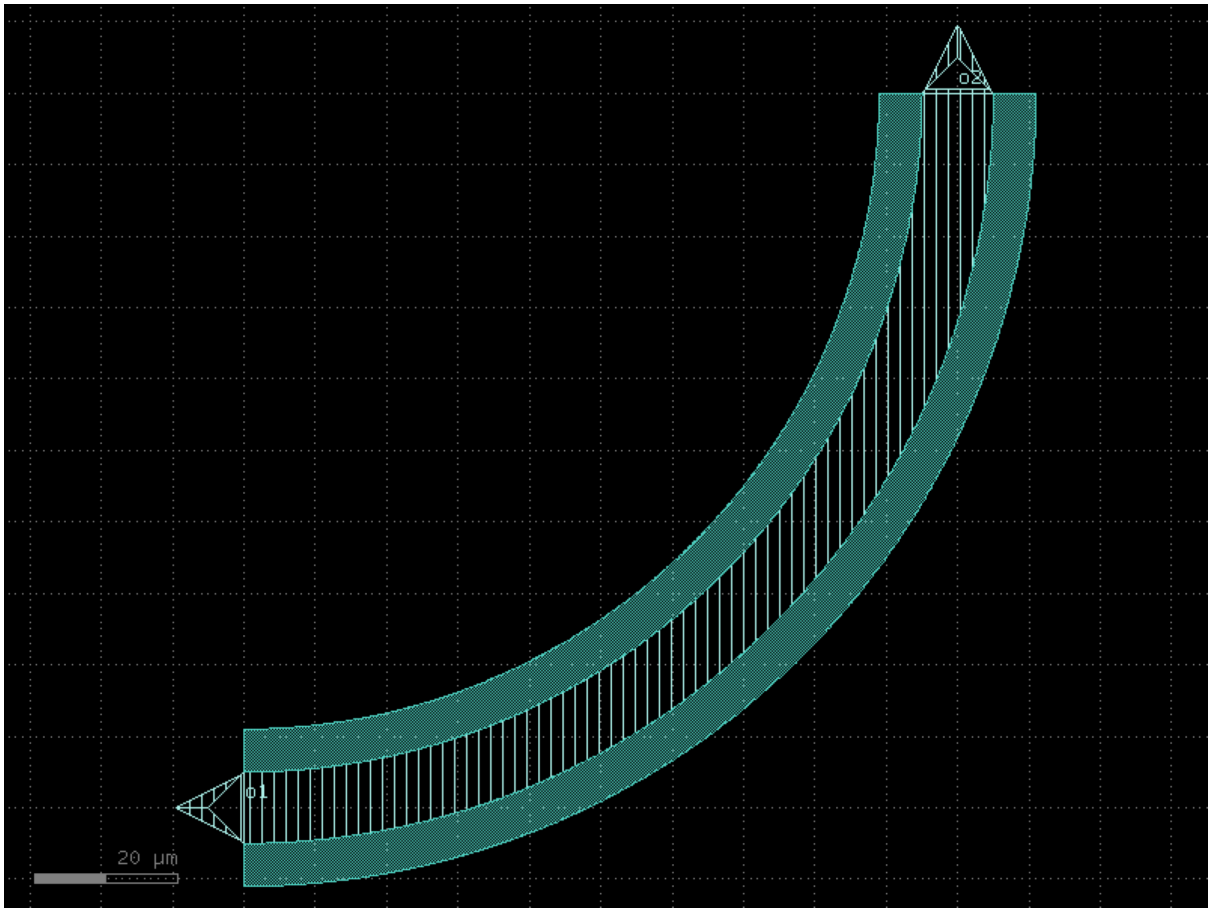
****kwargs** (*Unpack[BendCircularAllAngleKwargs]*) – Arguments passed to gf.c.bend_circular_all_angle.

Return type

ComponentAllAngle

```
from qpdk import cells, PDK

PDK.activate()
c = cells.bend_circular_all_angle().copy()
c.draw_ports()
c.plot()
```



1.1.4 bend_euler

qpdk.cells.**bend_euler**(***kwargs*)

Regular degree euler bend.

Parameters

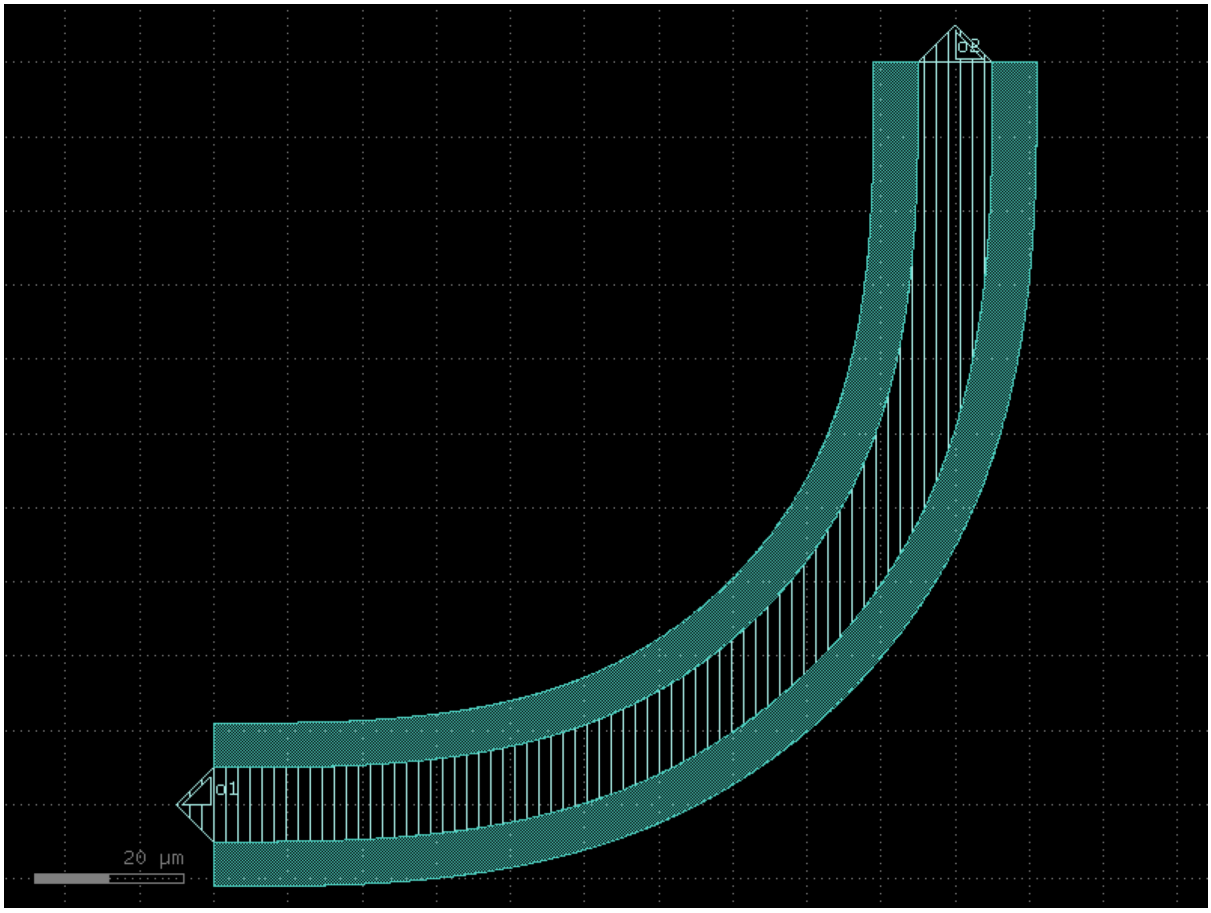
****kwargs** (*Unpack[BendEulerKwargs]*) – Arguments passed to gf.c.bend_euler.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.bend_euler().copy()
c.draw_ports()
c.plot()
```



1.1.5 bend_euler_all_angle

qpdk.cells.**bend_euler_all_angle**(***kwargs*)

Returns regular degree euler bend with arbitrary angle.

Parameters

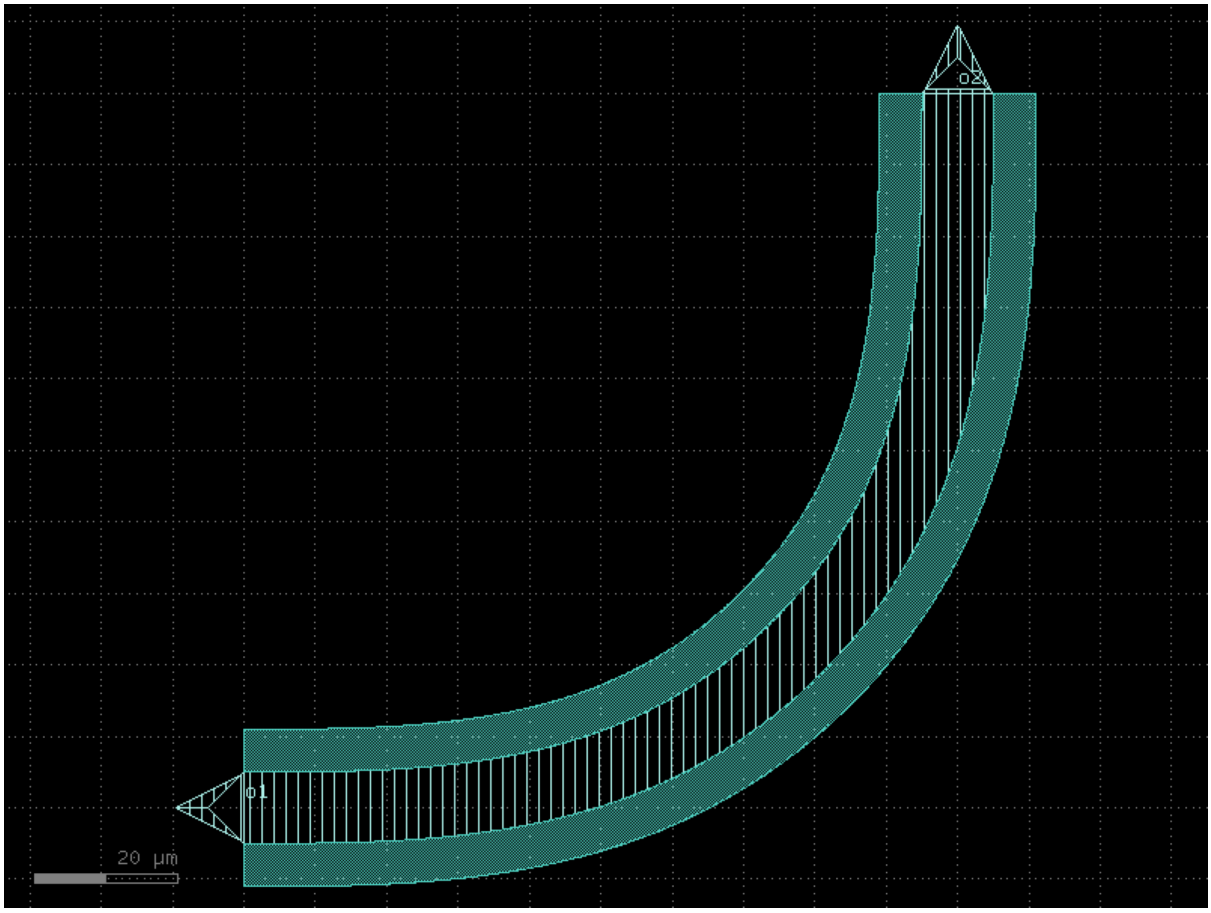
****kwargs** (*Unpack[BendEulerAllAngleKwargs]*) – Arguments passed to gf.c.bend_euler_all_angle.

Return type

ComponentAllAngle

```
from qpdk import cells, PDK

PDK.activate()
c = cells.bend_euler_all_angle().copy()
c.draw_ports()
c.plot()
```



1.1.6 bend_s

qpdk.cells.**bend_s**(***kwargs*)

Return S bend with bezier curve.

stores min_bend_radius property in self.info['min_bend_radius'] min_bend_radius depends on height and length

Parameters

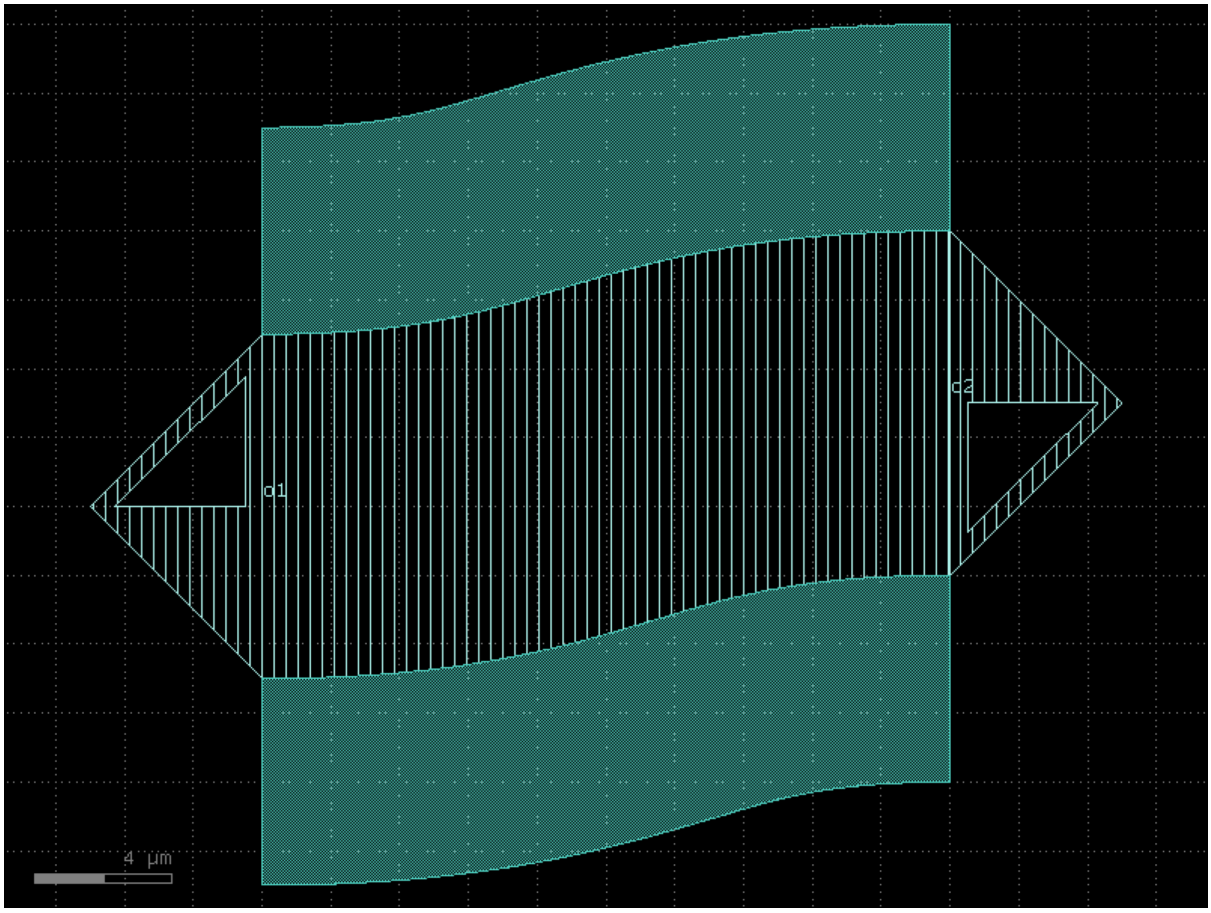
****kwargs** (*Unpack[BendSKwargs]*) – Arguments passed to gf.c.bend_s.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.bend_s().copy()
c.draw_ports()
c.plot()
```



1.1.7 circle

qpdk.cells.**circle**(*radius=10.0*, *angle_resolution=2.5*, *layer='WG'*)

Generate a circle geometry.

Parameters

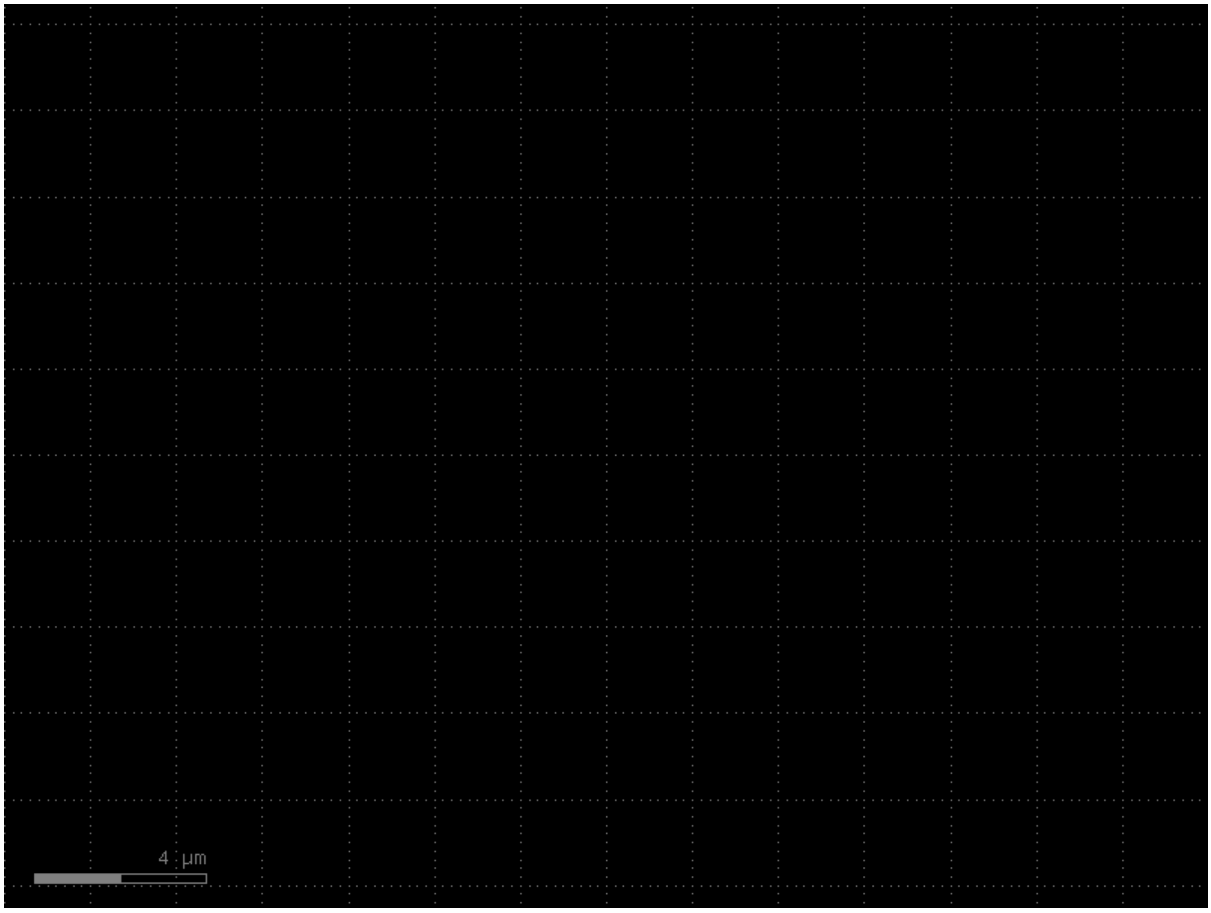
- **radius** (*float*) – of the circle.
- **angle_resolution** (*float*) – number of degrees per point.
- **layer** (*LayerSpec*) – layer.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.circle(radius=10, angle_resolution=2.5, layer='WG').copy()
c.draw_ports()
c.plot()
```



1.1.8 coupler_tunable

qpdk.cells.**coupler_tunable**(*pad_width=30.0*, *pad_height=40.0*, *gap=3.0*, *tuning_pad_width=15.0*, *tuning_pad_height=20.0*, *tuning_gap=1.0*, *feed_width=10.0*, *feed_length=30.0*, *layer_metal=<LayerMapQPDK.M1_DRAW: 1>*, *port_type='electrical'*)

Creates a tunable capacitive coupler with voltage control.

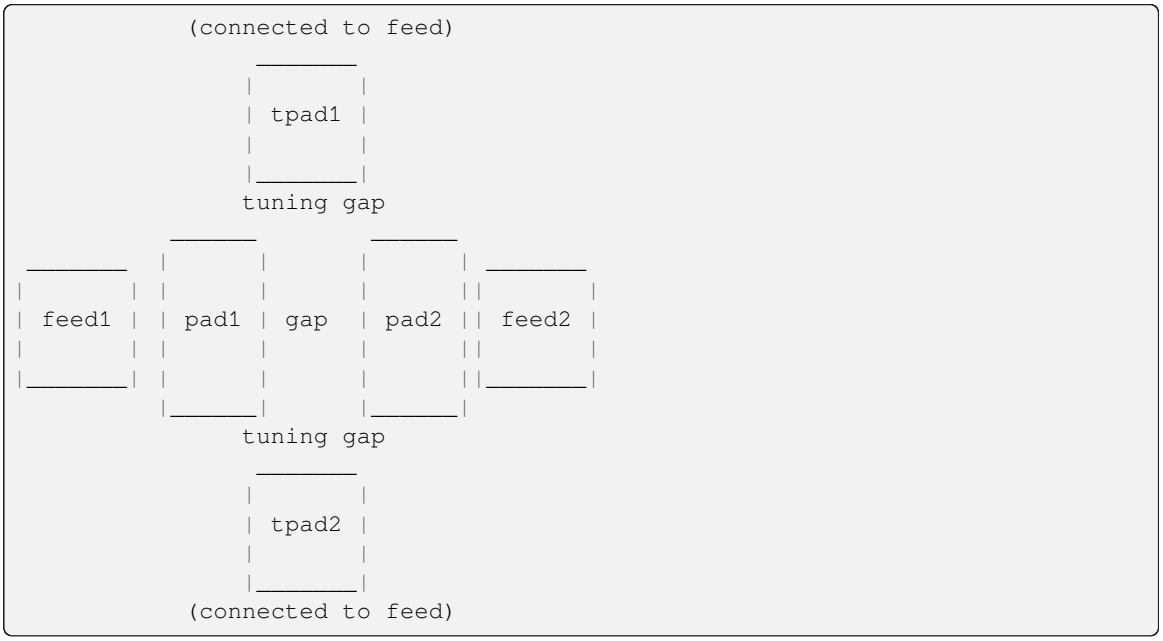
A tunable coupler includes additional electrodes that can be voltage-biased to change the coupling strength dynamically.

Parameters

- **pad_width** (*float*) – Width of main coupling pads in µm.
- **pad_height** (*float*) – Height of main coupling pads in µm.
- **gap** (*float*) – Gap between main coupling pads in µm.
- **tuning_pad_width** (*float*) – Width of tuning pads in µm.
- **tuning_pad_height** (*float*) – Height of tuning pads in µm.
- **tuning_gap** (*float*) – Gap to tuning pads in µm.
- **feed_width** (*float*) – Width of feed lines in µm.
- **feed_length** (*float*) – Length of feed lines in µm.
- **layer_metal** (*LayerSpec*) – Layer for main metal structures.
- **port_type** (*str*) – Type of port to add to the component.

Returns
A gdsfactory component with the tunable coupler geometry.

Return type
Component




```
from qpdk import cells, PDK

PDK.activate()
c = cells.coupler_tunable(pad_width=30, pad_height=40, gap=3, tuning_pad_width=15,␣
↪tuning_pad_height=20, tuning_gap=1, feed_width=10, feed_length=30, layer_metal=
↪'M1_DRAW', port_type='electrical').copy()
c.draw_ports()
c.plot()
```

1.1.9 double_pad_transmon

qpdk.cells.**double_pad_transmon**(***kwargs*)

Creates a double capacitor pad transmon qubit with Josephson junction.

A transmon qubit consists of two capacitor pads connected by a Josephson junction. The junction creates an anharmonic oscillator that can be used as a qubit.

See [KYG+07] for details.

Parameters
****kwargs** (*Unpack[DoublePadTransmonParams]*) – DoublePadTransmon-Params for the transmon qubit.

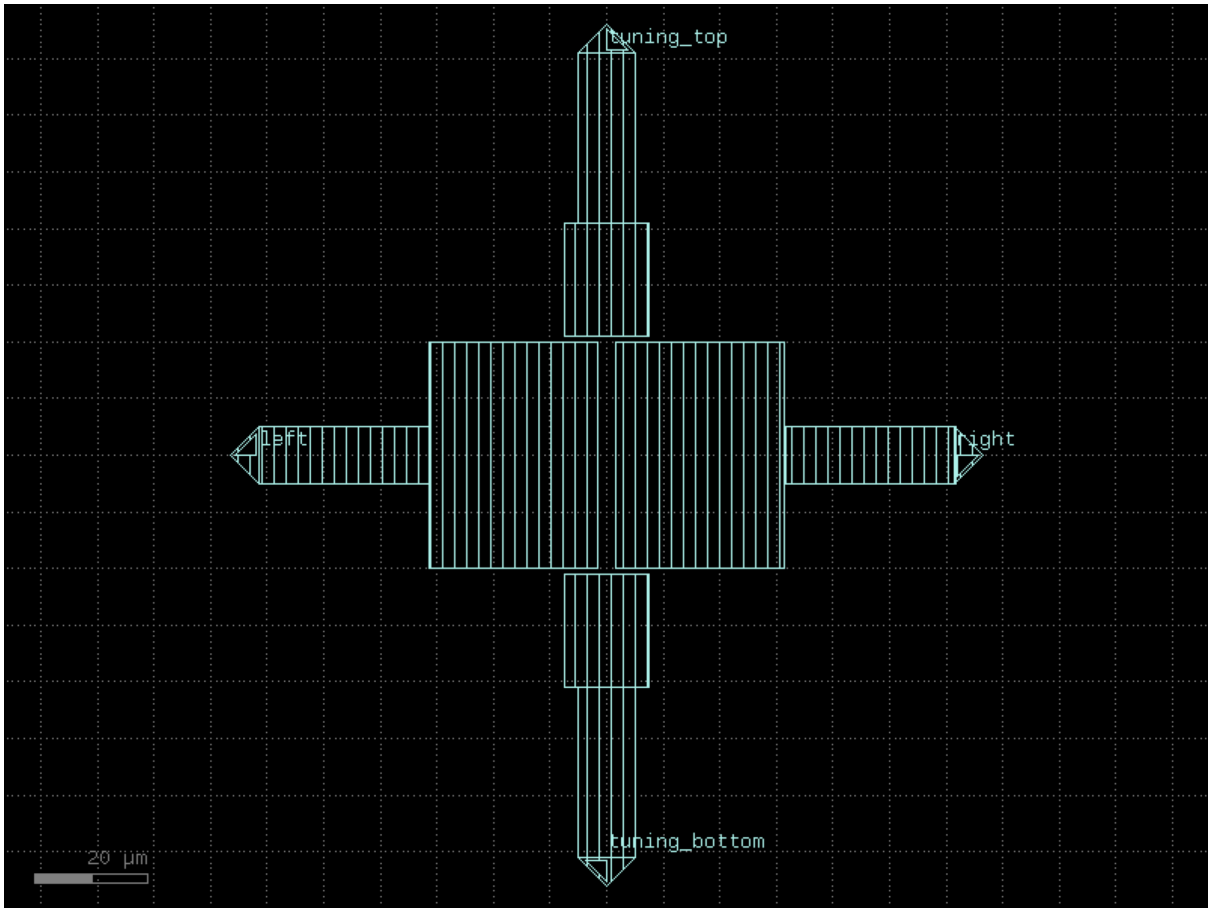
Returns
A gdsfactory component with the transmon geometry.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.double_pad_transmon().copy()
```

(continues on next page)



(continued from previous page)

```
c.draw_ports()
c.plot()
```

1.1.10 double_pad_transmon_with_bbox

qpdk.cells.**double_pad_transmon_with_bbox**(*bbox_extension=200.0*, ***kwargs*)

Creates a double capacitor pad transmon qubit with Josephson junction and an etched bounding box.

See *double_pad_transmon()* (page 13) for more details.

Parameters

- **bbox_extension** (*float*) – Extension size for the bounding box in µm.
- ****kwargs** (*Unpack[DoublePadTransmonParams]*) – DoublePadTransmonParams for the transmon qubit.

Returns

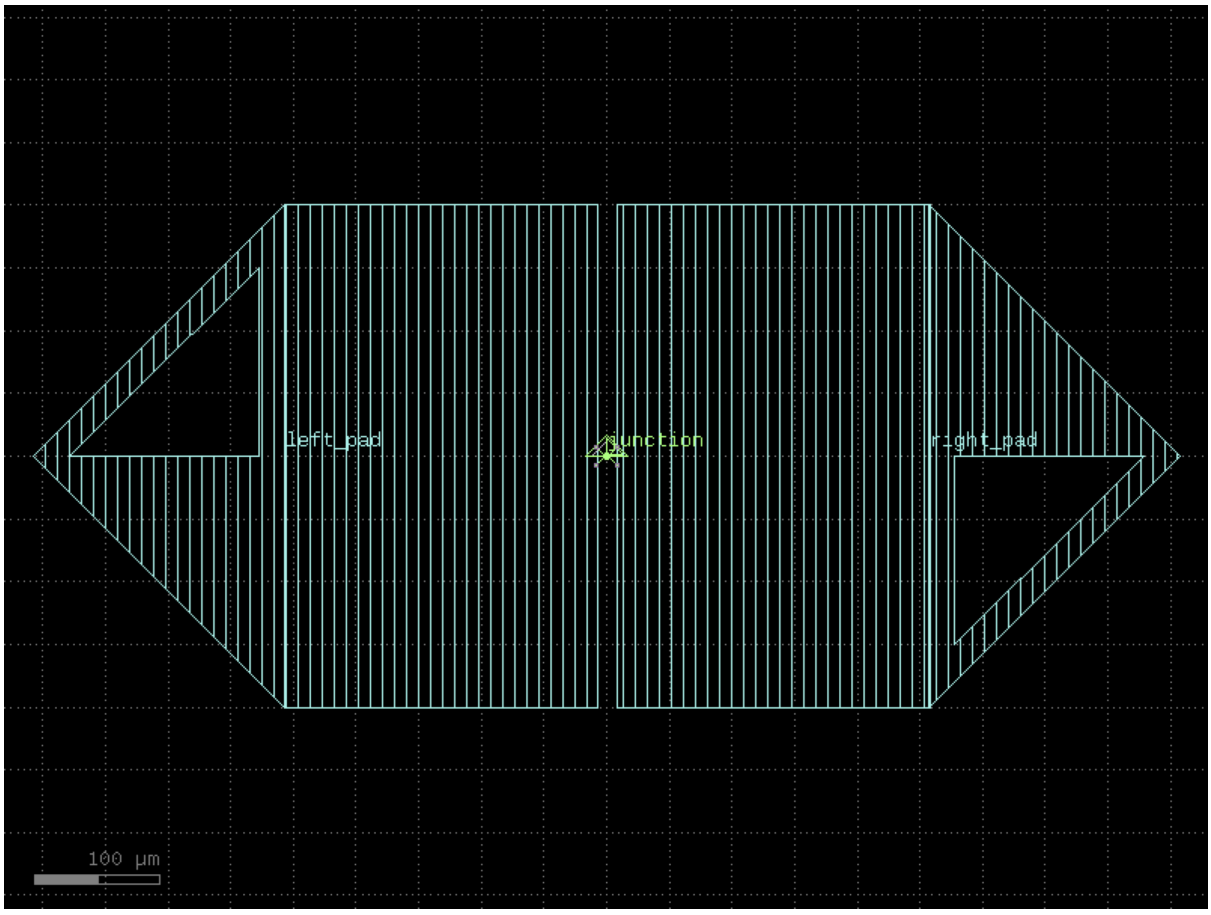
A gdsfactory component with the transmon geometry and etched box.

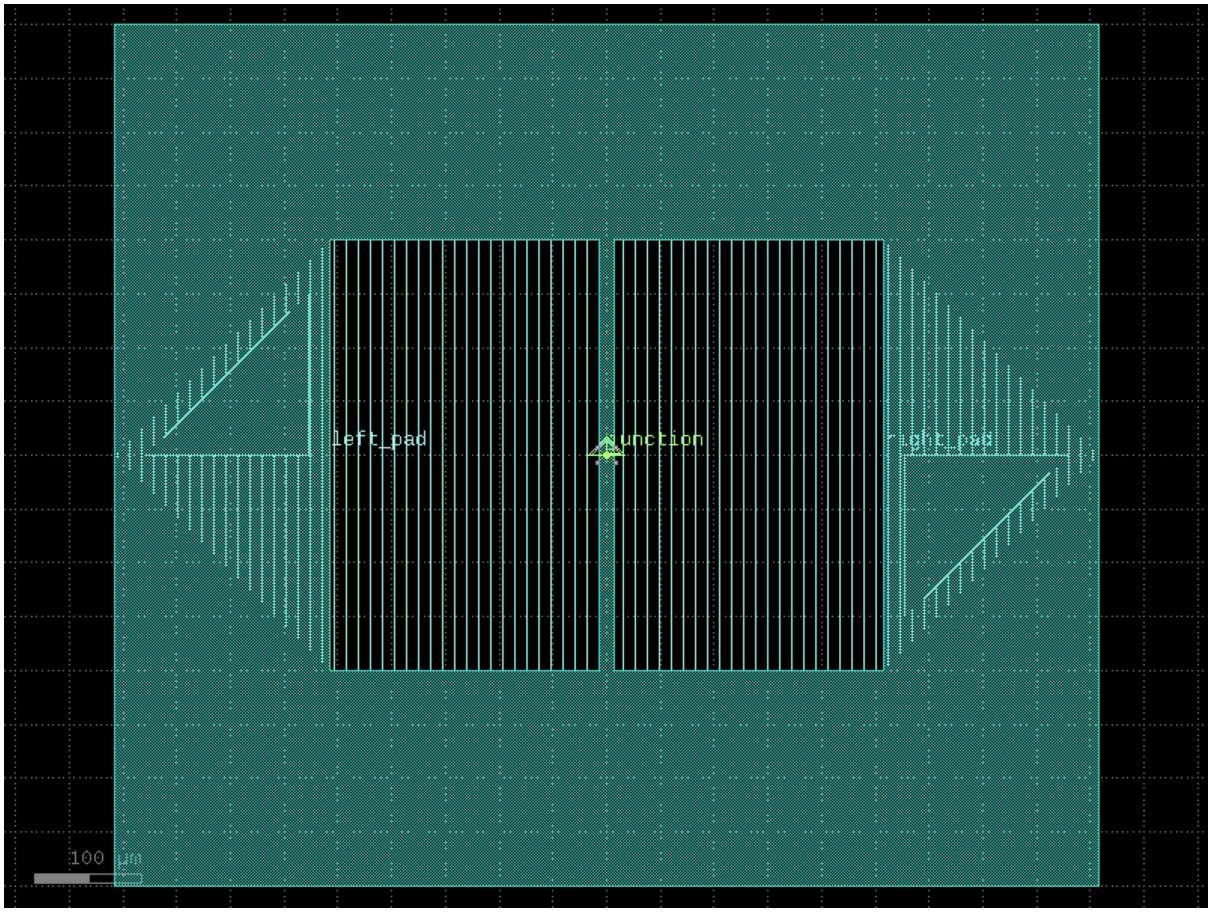
Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.double_pad_transmon_with_bbox(bbox_extension=200).copy()
c.draw_ports()
c.plot()
```


left_pad
junction
right_pad
100 µm


left_pad
junction
right_pad
100 µm

1.1.11 flipmon

qpdk.cells.**flipmon**(***kwargs*)

Creates a circular transmon qubit with *flipmon* geometry.

A circular variant of the transmon qubit with another circle as the inner pad.

See [LWJ+25, LZY+21] for details about the *flipmon* design.

Parameters

****kwargs** (*Unpack[FlipmonParams]*) – FlipmonParams for the flipmon qubit.

Keyword Arguments

- **inner_ring_radius** – Central radius of the inner circular capacitor pad in μm.
- **inner_ring_width** – Width of the inner circular capacitor pad in μm.
- **outer_ring_radius** – Central radius of the outer circular capacitor pad in μm.
- **outer_ring_width** – Width of the outer circular capacitor pad in μm.
- **top_circle_radius** – Central radius of the top circular capacitor pad in μm. There is no separate width as the filled circle is not a ring.
- **junction_spec** – Component specification for the Josephson junction component.
- **junction_displacement** – Optional complex transformation to apply to the junction.
- **layer_metal** – Layer for the metal pads.
- **layer_metal_top** – Layer for the other metal layer pad for flip-chip.

Returns

A gdsfactory component with the circular transmon geometry.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.flipmon().copy()
c.draw_ports()
c.plot()
```

1.1.12 flipmon_with_bbox

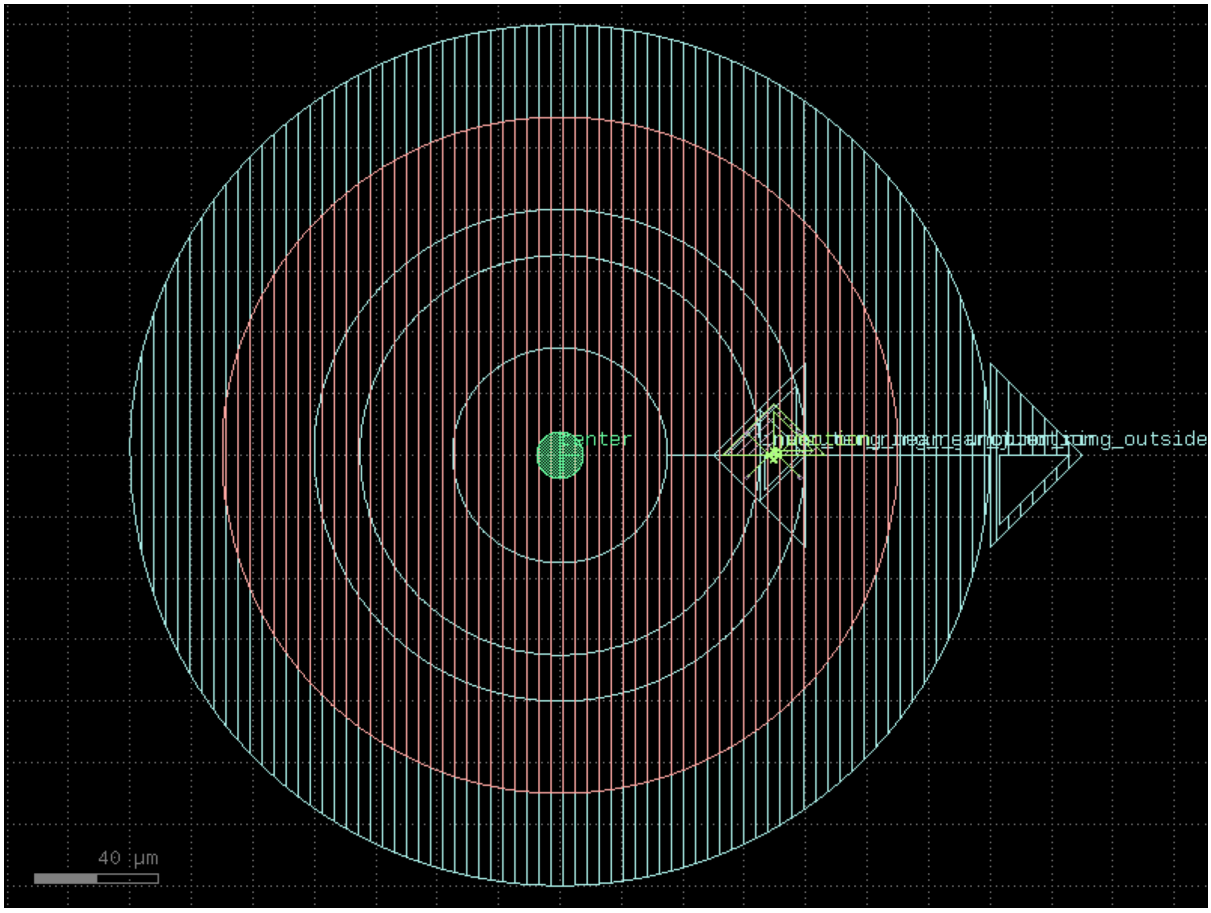
qpdk.cells.**flipmon_with_bbox**(*flipmon_params=None*, *m1_etch_extension_gap=30.0*, *m2_etch_extension_gap=40.0*)

Creates a circular transmon qubit with *flipmon* geometry and a circular etched bounding box.

See *flipmon()* (page 16) for more details.

Parameters

- **flipmon_params** (*FlipmonParams | None*) – FlipmonParams for the flipmon qubit.
- **m1_etch_extension_gap** (*float*) – Radius extension length for the M1 etch bounding box in μm.
- **m2_etch_extension_gap** (*float*) – Radius extension length for the M2 etch bounding box in μm.



Returns
A gdsfactory component with the flipmon geometry and etched box.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.flipmon_with_bbox(m1_etch_extension_gap=30, m2_etch_extension_gap=40).
 ↪copy()
c.draw_ports()
c.plot()
```

1.1.13 indium_bump

qpdk.cells.**indium_bump**(*diameter=15.0*)

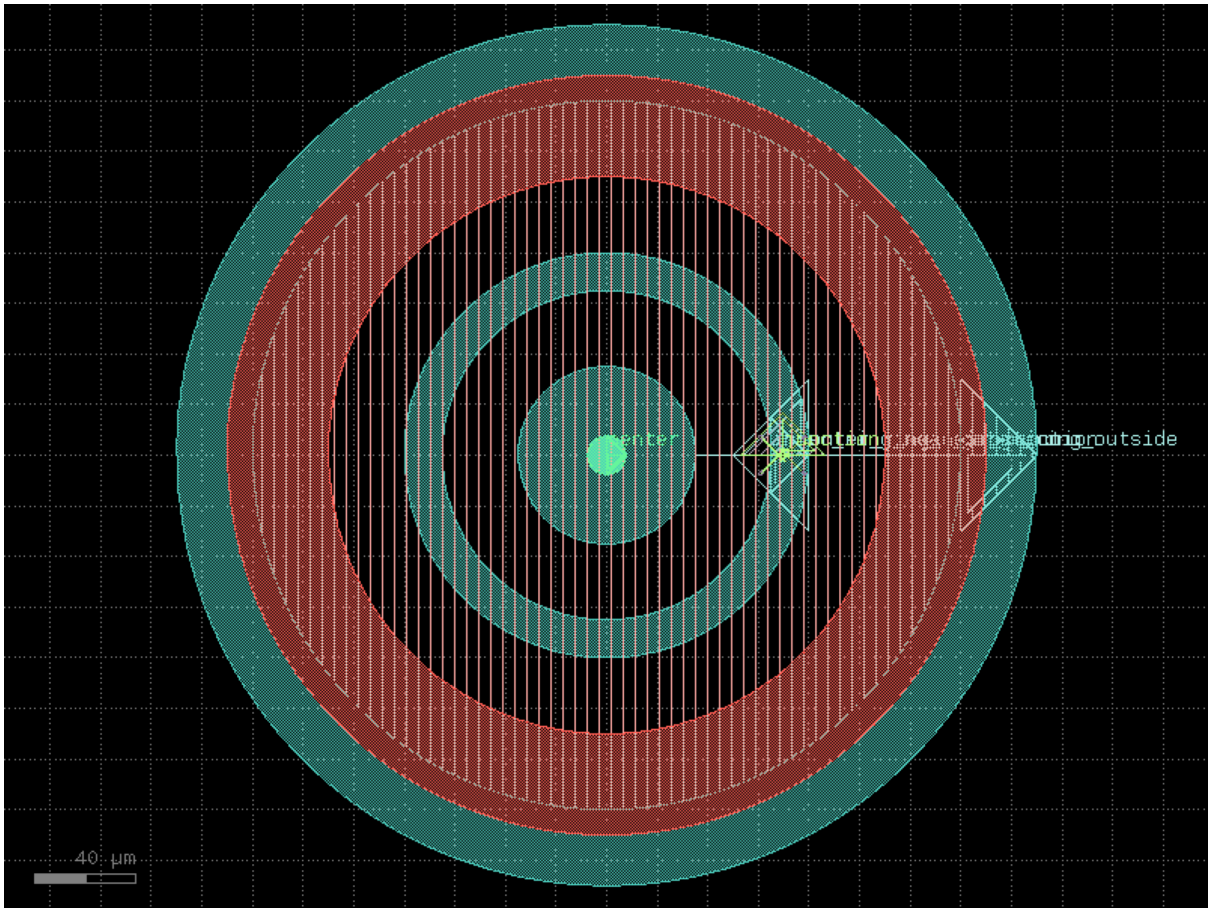
Creates an indium bump component for 3D integration.

See [RKD+17] for details.

Parameters
diameter (*float*) – Diameter of the indium bump in µm.

Returns
A gdsfactory Component representing the indium bump.

Return type
Component




```
from qpdk import cells, PDK

PDK.activate()
c = cells.indium_bump(diameter=15).copy()
c.draw_ports()
c.plot()
```

1.1.14 interdigital_capacitor

qpdk.cells.**interdigital_capacitor**(***kwargs*)

Generate an interdigital capacitor component with ports on both ends.

An interdigital capacitor consists of interleaved metal fingers that create a distributed capacitance. This component creates a planar capacitor with two sets of interleaved fingers extending from opposite ends.

See for example [LeiZhuW00].

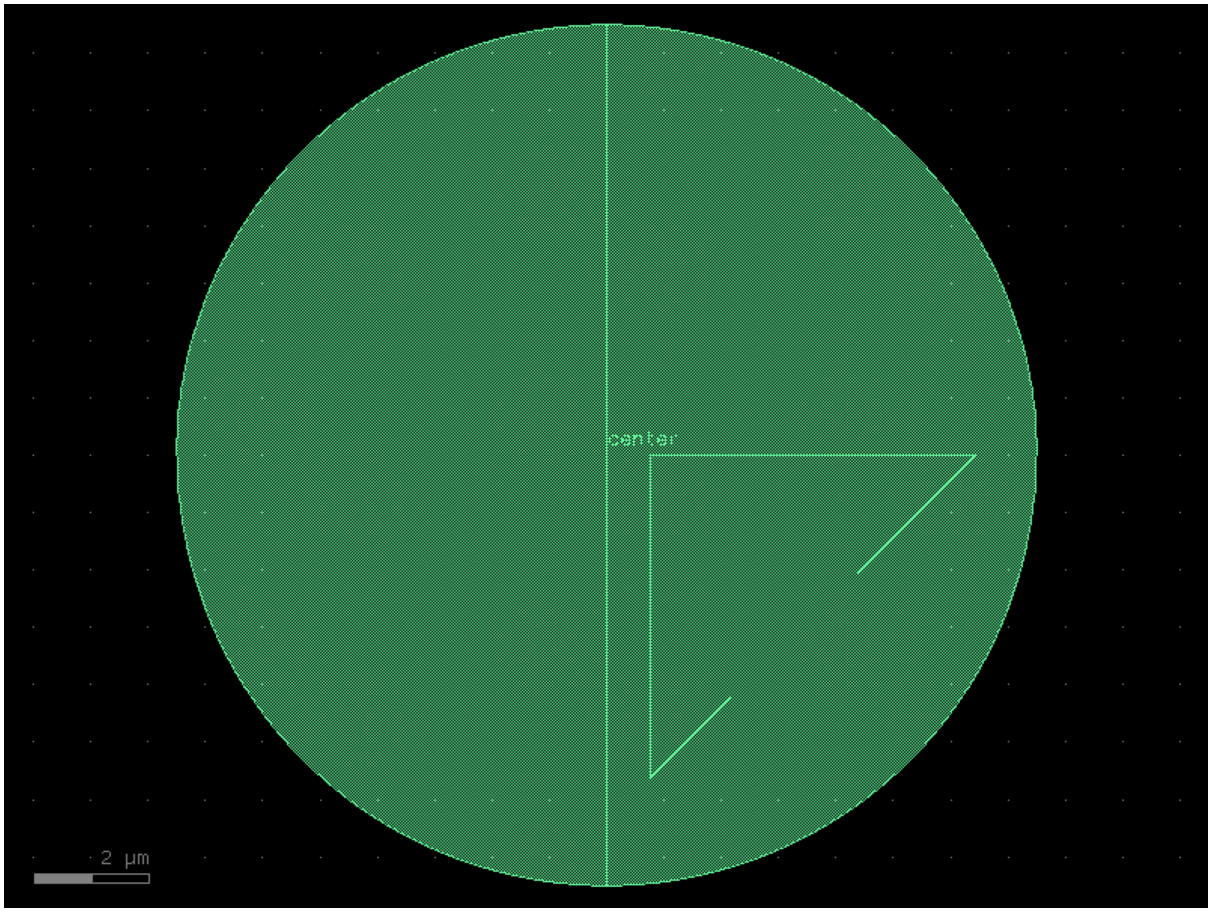
Note

`finger_length=0` effectively provides a parallel plate capacitor. The capacitance scales approximately linearly with the number of fingers and finger length.

Parameters

kwargs (*Unpack[InterdigitalCapacitorParams]*) – `InterdigitalCapacitorParams` for the interdigital capacitor.

Returns



A gdsfactory component with the interdigital capacitor geometry
and two ports (‘o1’ and ‘o2’) on opposing sides.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.interdigital_capacitor().copy()
c.draw_ports()
c.plot()
```

1.1.15 josephson_junction

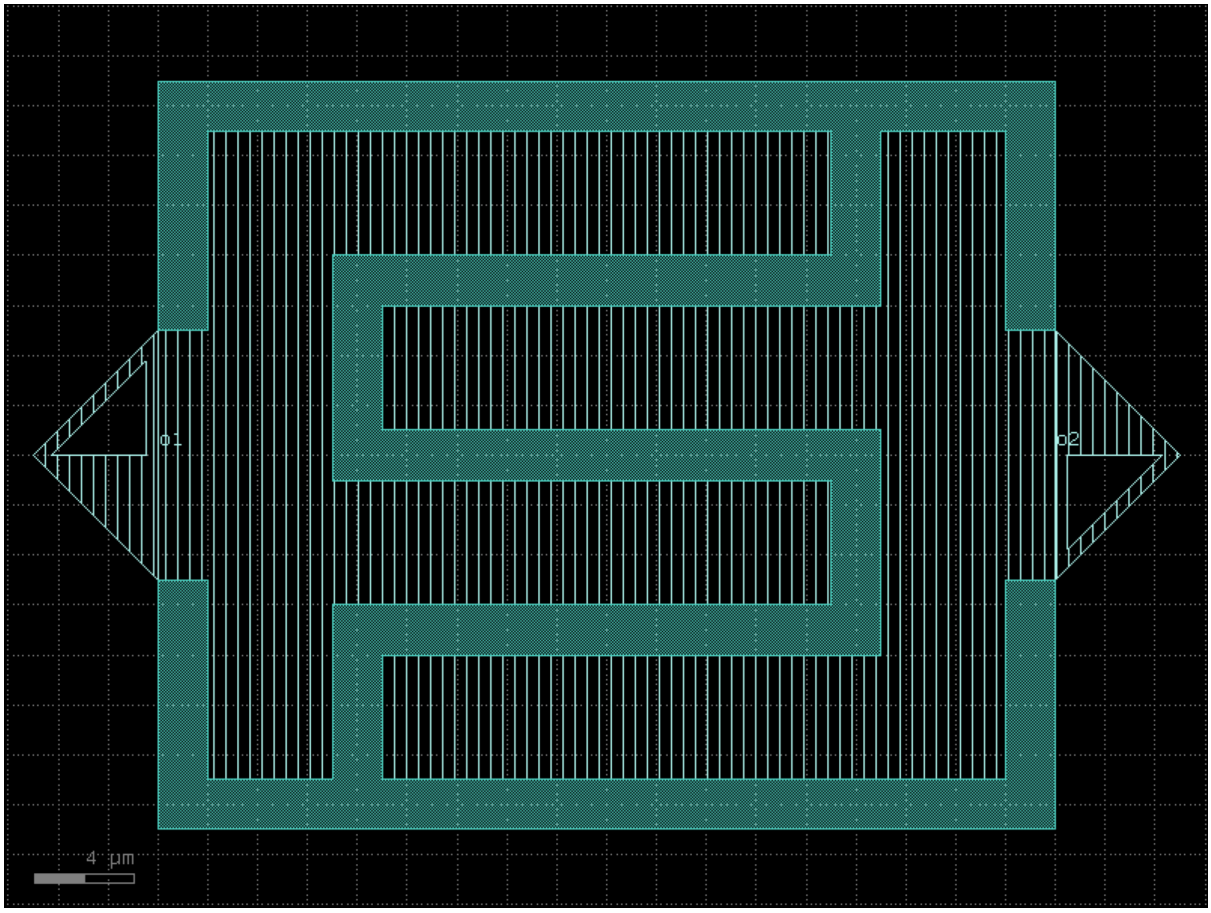
qpdk.cells.**josephson_junction**(*junction_overlap_displacement=1.8*, ***kwargs*)

Creates a single Josephson junction component.

A Josephson junction consists of two superconducting electrodes separated by a thin insulating barrier allowing tunnelling.

Parameters

- **junction_overlap_displacement** (*float*) – Displacement of the overlap region in µm. Measured from the centers of the junction ports
- **kwargs** (*Unpack[SingleJosephsonJunctionWireParams]*) – SingleJosephsonJunctionWireParams for single wires.



Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.josephson_junction(junction_overlap_displacement=1.8).copy()
c.draw_ports()
c.plot()
```

1.1.16 launcher

qpdk.cells.**launcher**(*straight_length=200.0*, *taper_length=100.0*, *cross_section_big=<function launcher_cross_section_big>*, *cross_section_small='cpw'*)

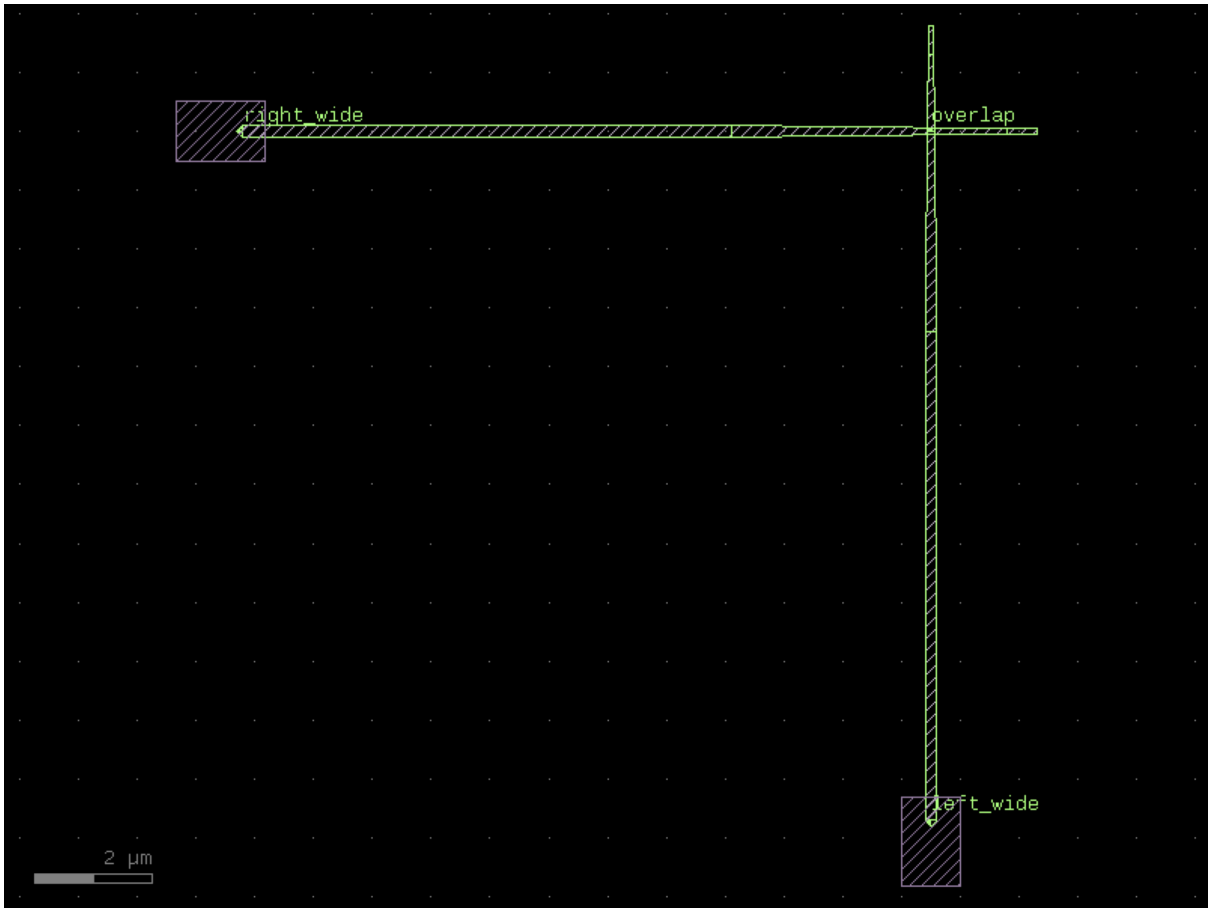
Generate an RF launcher pad for wirebonding or probe testing.

Creates a launcher component consisting of a straight section with large cross-section connected to a tapered transition down to a smaller cross-section. This design facilitates RF signal access through probes or wirebonds while maintaining good impedance matching.

The default dimensions are taken from [TSKivijarvi+25].

Parameters

- **straight_length** (*float*) – Length of the straight, wirebond landing area, section in µm.
- **taper_length** (*float*) – Length of the taper section in µm.



- **cross_section_big** (*CrossSectionSpec*) – Cross-section specification for the large end of the launcher (probe/wirebond interface).
- **cross_section_small** (*CrossSectionSpec*) – Cross-section specification for the small end of the launcher (circuit interface).

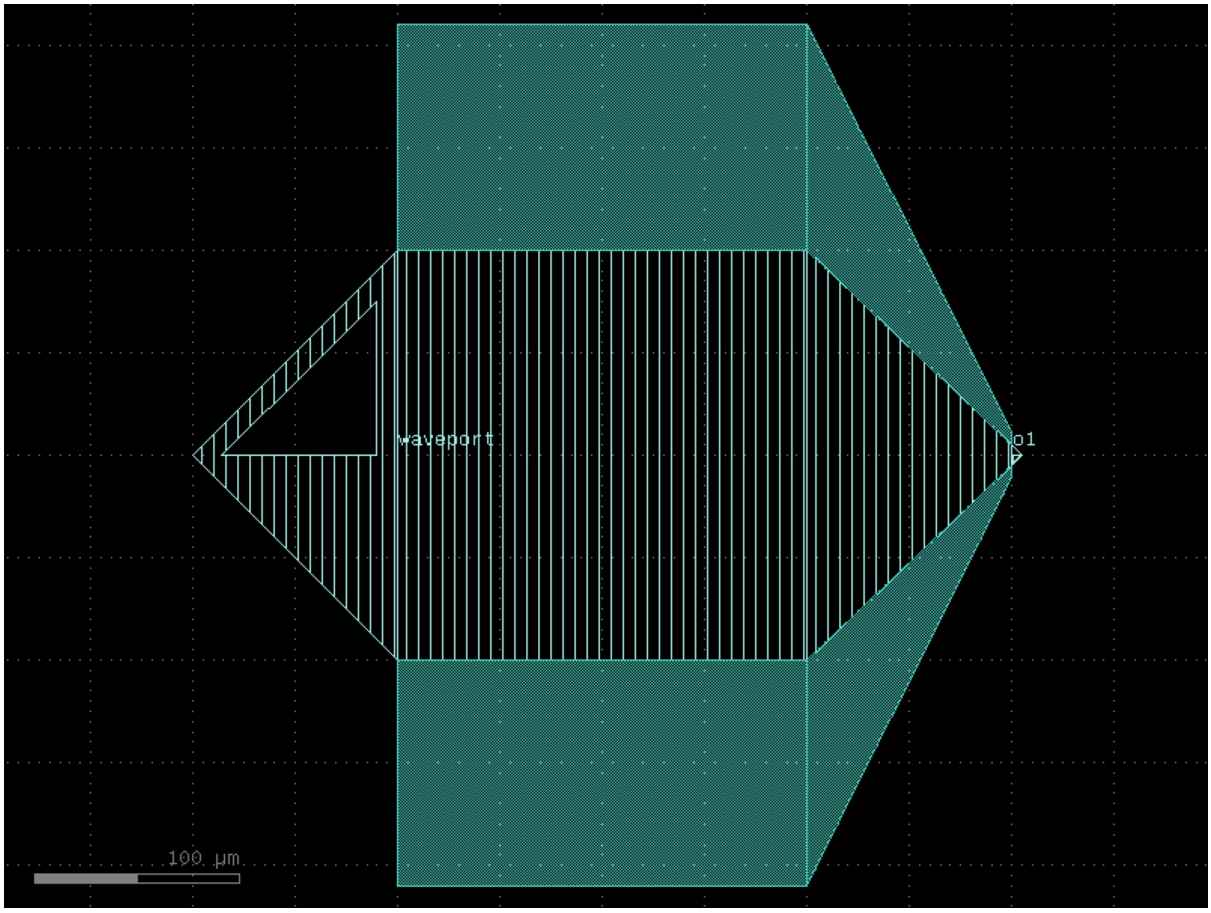
Returns

A gdsfactory component containing the complete launcher
geometry with one output port ("o1") at the small end.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.launcher(straight_length=200, taper_length=100, cross_section_small='cpw
↪').copy()
c.draw_ports()
c.plot()
```



1.1.17 nxn

qpdk.cells.**nxn**(***kwargs*)

Returns a tee waveguide.

Parameters

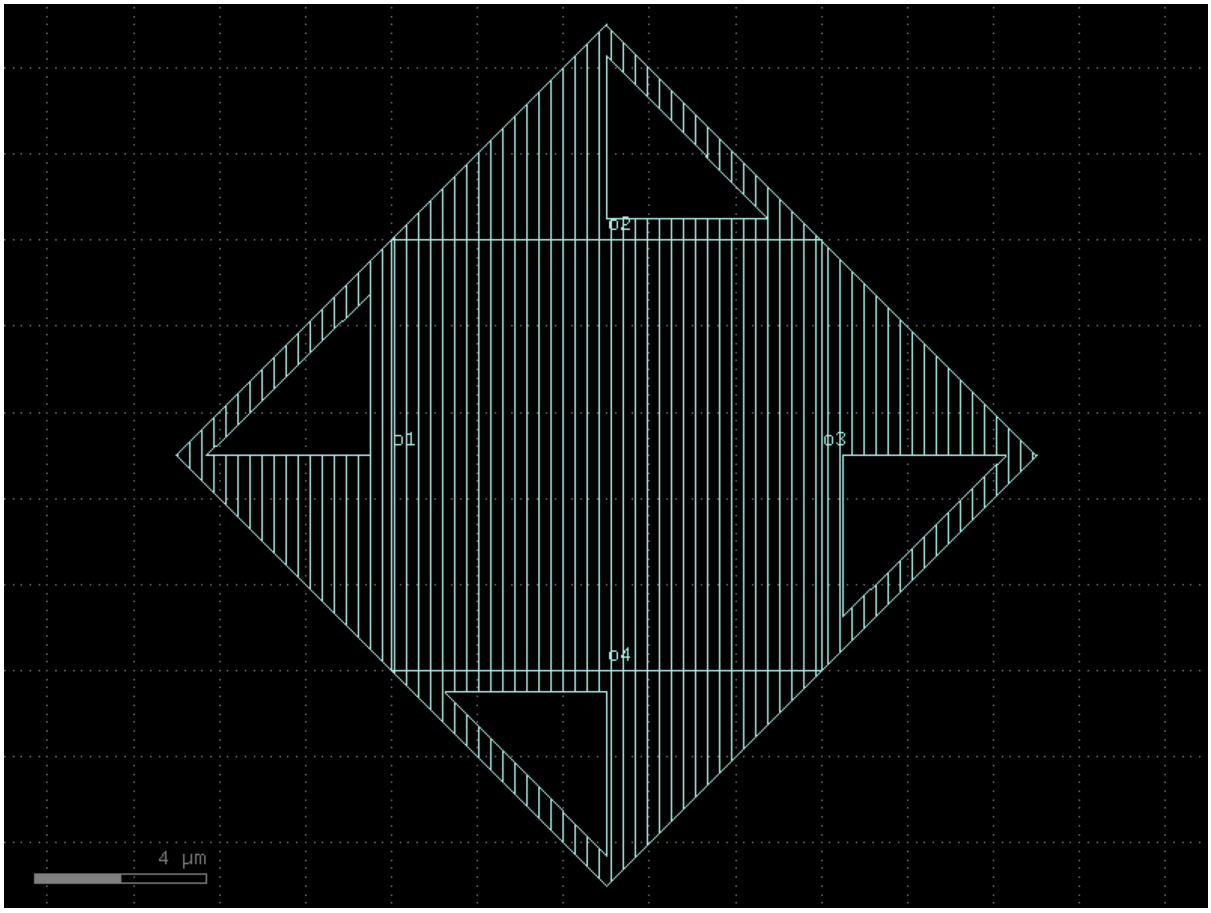
****kwargs** (*Unpack[NxnKwargs]*) – Arguments passed to gf.c.nxn.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.nxn().copy()
c.draw_ports()
c.plot()
```



1.1.18 plate_capacitor

qpdk.cells.**plate_capacitor**(***kwargs*)

Creates a plate capacitor.

A capacitive coupler consists of two metal pads separated by a small gap, providing capacitive coupling between circuit elements like qubits and resonators.

> **Note**
>
> This is a special case of the interdigital capacitor with zero finger length.

Parameters

****kwargs** (*Unpack[InterdigitalCapacitorParams]*) – InterdigitalCapacitorParams for the interdigital

Returns

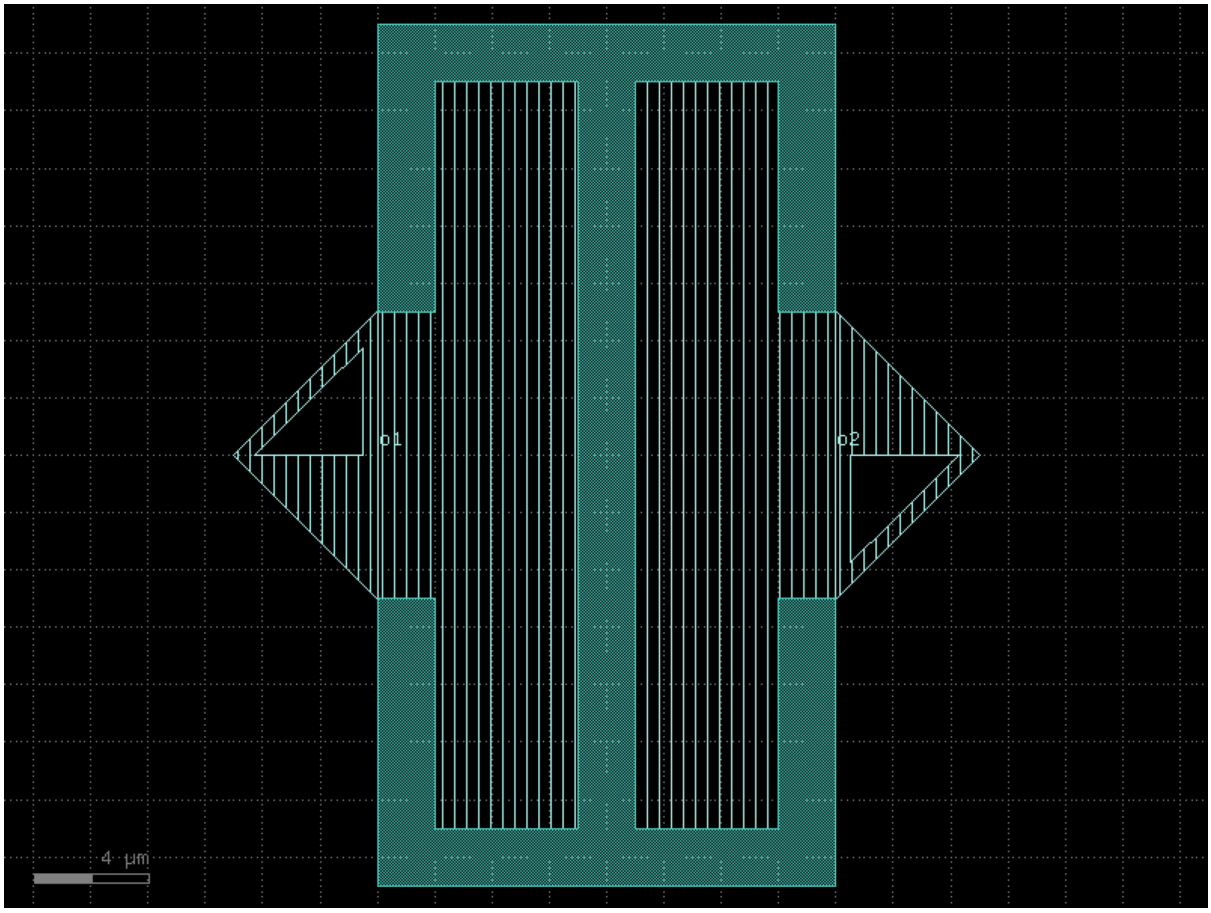
A gdsfactory component with the plate capacitor geometry.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.plate_capacitor().copy()
c.draw_ports()
c.plot()
```



1.1.19 plate_capacitor_single

qpdk.cells.**plate_capacitor_single**(***kwargs*)

Creates a single plate capacitor for coupling.

This is essentially half of a `plate capacitor()`.

Parameters

****kwargs** (*Unpack[InterdigitalCapacitorParams]*) – InterdigitalCapacitorParams

Returns

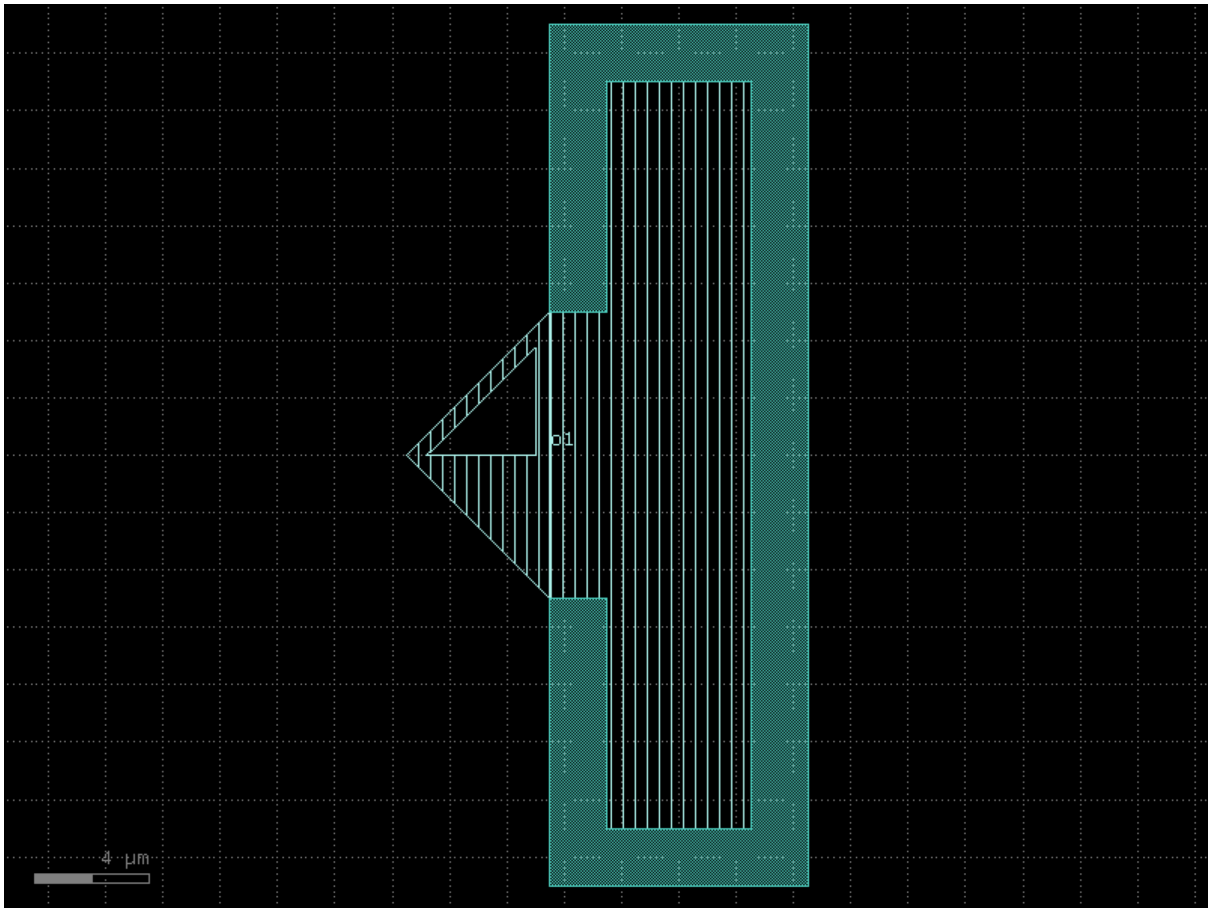
A gdsfactory component with the plate capacitor geometry.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.plate_capacitor_single().copy()
c.draw_ports()
c.plot()
```



1.1.20 rectangle

qpdk.cells.**rectangle**(*size=(4.0, 2.0)*, *layer='M1_DRAW'*, *centered=False*, *port_type='electrical'*, *port_orientations=(180, 90, 0, -90)*)

Returns a rectangle.

Parameters

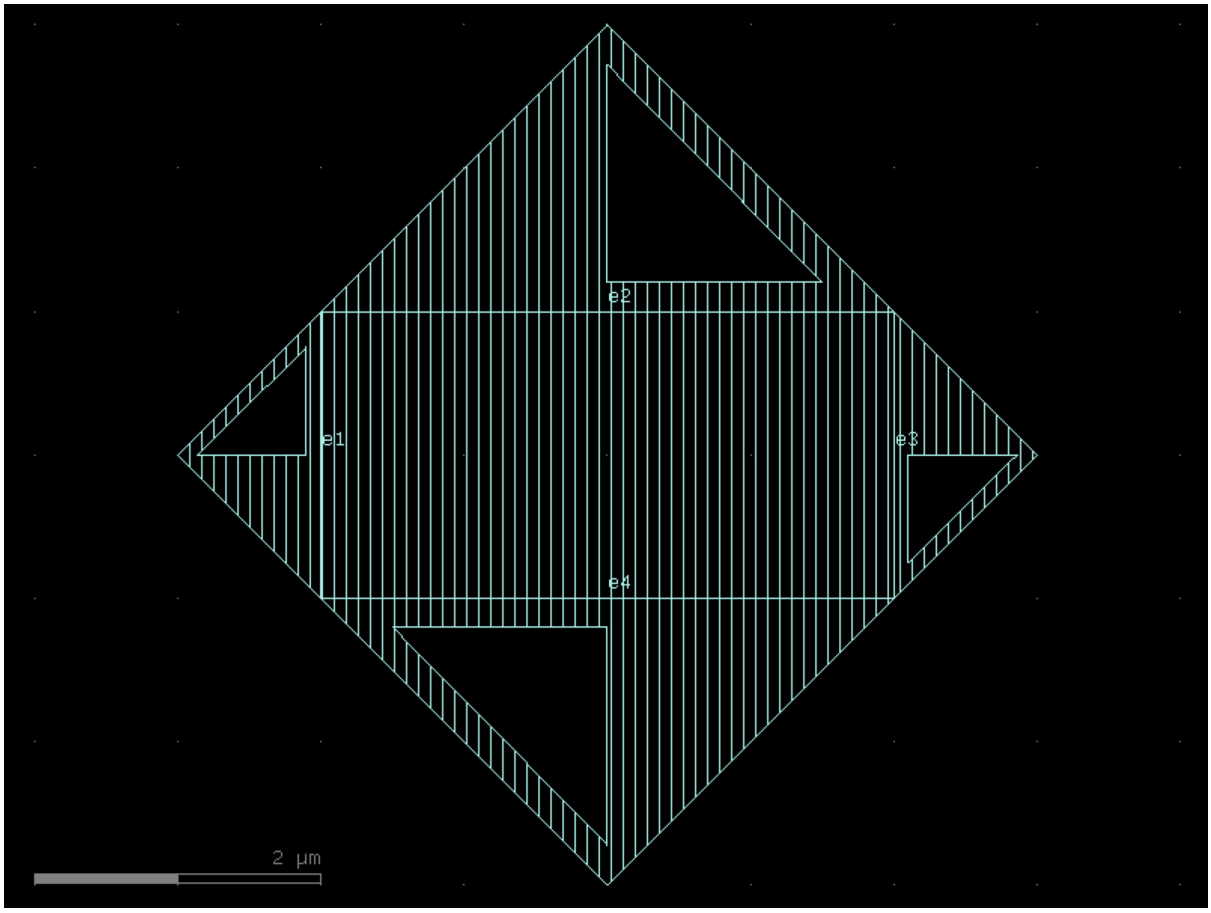
- **size** (*tuple[float, float]*) – (tuple) Width and height of rectangle.
- **layer** (*tuple[int, int] | str | int | LayerEnum*) – Specific layer to put polygon geometry on.
- **centered** (*bool*) – True sets center to (0, 0), False sets south-west to (0, 0).
- **port_type** (*str | None*) – optical, electrical.
- **port_orientations** (*tuple[int, ...] | list[int] | None*) – list of port_orientations to add. None adds no ports.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.rectangle(size=(4, 2), layer='M1_DRAW', centered=False, port_type=
 ↪'electrical', port_orientations=(180, 90, 0, -90)).copy()
c.draw_ports()
c.plot()
```



1.1.21 resonator

qpdk.cells.**resonator**(*length=4000.0*, *meanders=6*, *bend_spec=<function bend_circular>*, *cross_section='cpw'*, ***, *open_start=False*, *open_end=False*)

Creates a meandering coplanar waveguide resonator.

Changing *open_start* and *open_end* appropriately allows creating a shorted quarter-wave resonator or an open half-wave resonator.

See [MP12] for details

Parameters

- **length** (*float*) – Length of the resonator in µm.
- **meanders** (*int*) – Number of meander sections to fit the resonator in a compact area.
- **bend_spec** (*ComponentSpec*) – Specification for the bend component used in meanders.
- **cross_section** (*CrossSectionSpec*) – Cross-section specification for the resonator.
- **open_start** (*bool*) – If True, adds an etch section at the start of the resonator.
- **open_end** (*bool*) – If True, adds an etch section at the end of the resonator.

Returns

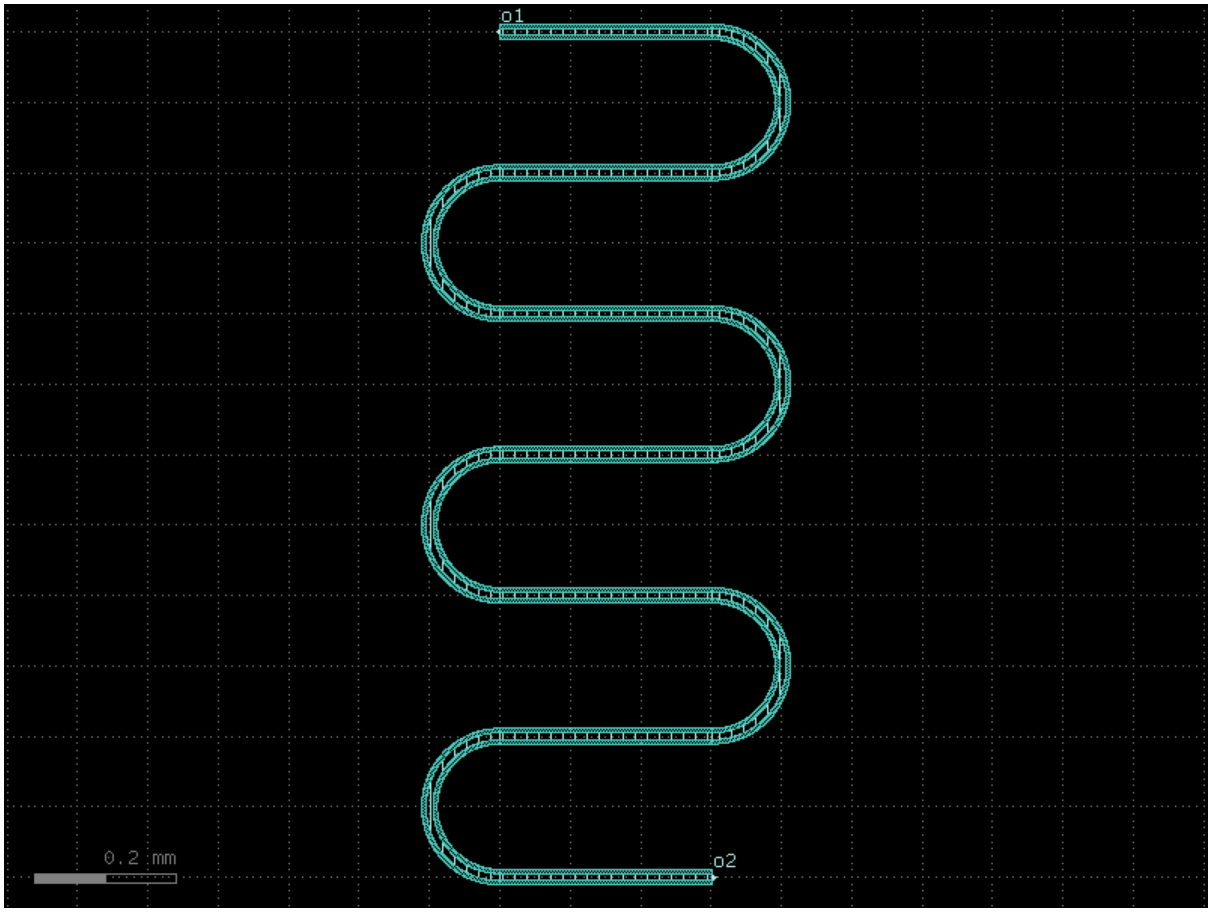
A gdsfactory component with meandering resonator geometry.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.resonator(length=4000, meanders=6, cross_section='cpw', open_start=False,
 ↪ open_end=False).copy()
c.draw_ports()
c.plot()
```



1.1.22 resonator_coupled

qpdk.cells.**resonator_coupled**(*resonator_params=None*, *cross_section_non_resonator='cpw'*, *coupling_straight_length=200.0*, *coupling_gap=12.0*)

Creates a meandering coplanar waveguide resonator with a coupling waveguide.

This component combines a resonator with a parallel coupling waveguide placed at a specified gap for proximity coupling. Similar to the design described in [BM18].

Parameters

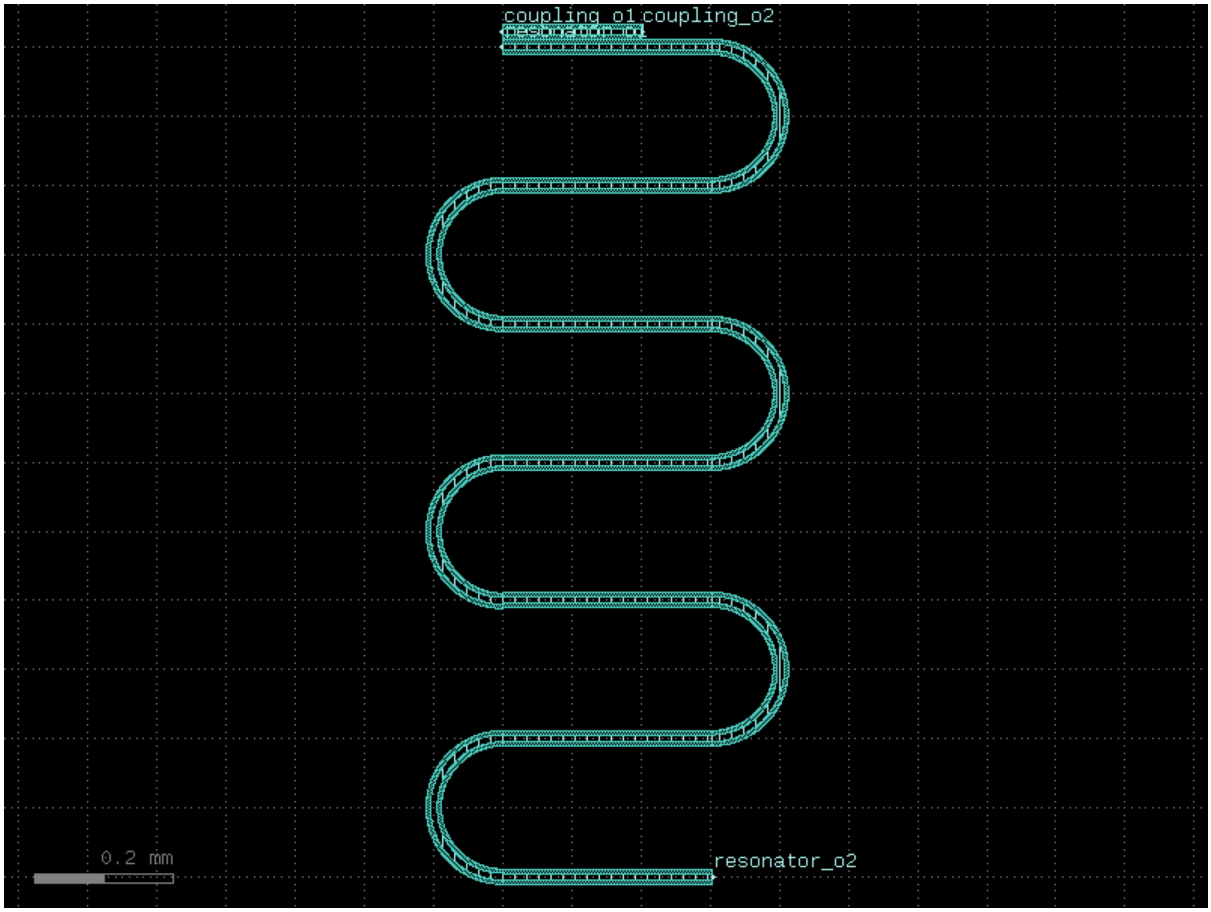
- **resonator_params** (*ResonatorParams | None*) – Parameters for the resonator component. If None, defaults will be used.
- **cross_section_non_resonator** (*CrossSectionSpec*) – Cross-section specification for the coupling waveguide.
- **coupling_straight_length** (*float*) – Length of the coupling waveguide section in µm.
- **coupling_gap** (*float*) – Gap between the resonator and coupling waveguide in µm. Measured from edges of the center conductors.

Returns
A gdsfactory component with meandering resonator and coupling waveguide.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.resonator_coupled(cross_section_non_resonator='cpw', coupling_straight_
↪length=200, coupling_gap=12).copy()
c.draw_ports()
c.plot()
```



1.1.23 resonator_half_wave

qpdk.cells.**resonator_half_wave**(*length=4000.0*, *meanders=6*, *bend_spec=<function bend_circular>*, *cross_section='cpw'*, ***, *open_start=True*, *open_end=True*)

Creates a meandering coplanar waveguide resonator.

Changing *open_start* and *open_end* appropriately allows creating a shorted quarter-wave resonator or an open half-wave resonator.

See [MP12] for details

Parameters

- **length** (*float*) – Length of the resonator in µm.
- **meanders** (*int*) – Number of meander sections to fit the resonator in a compact area.

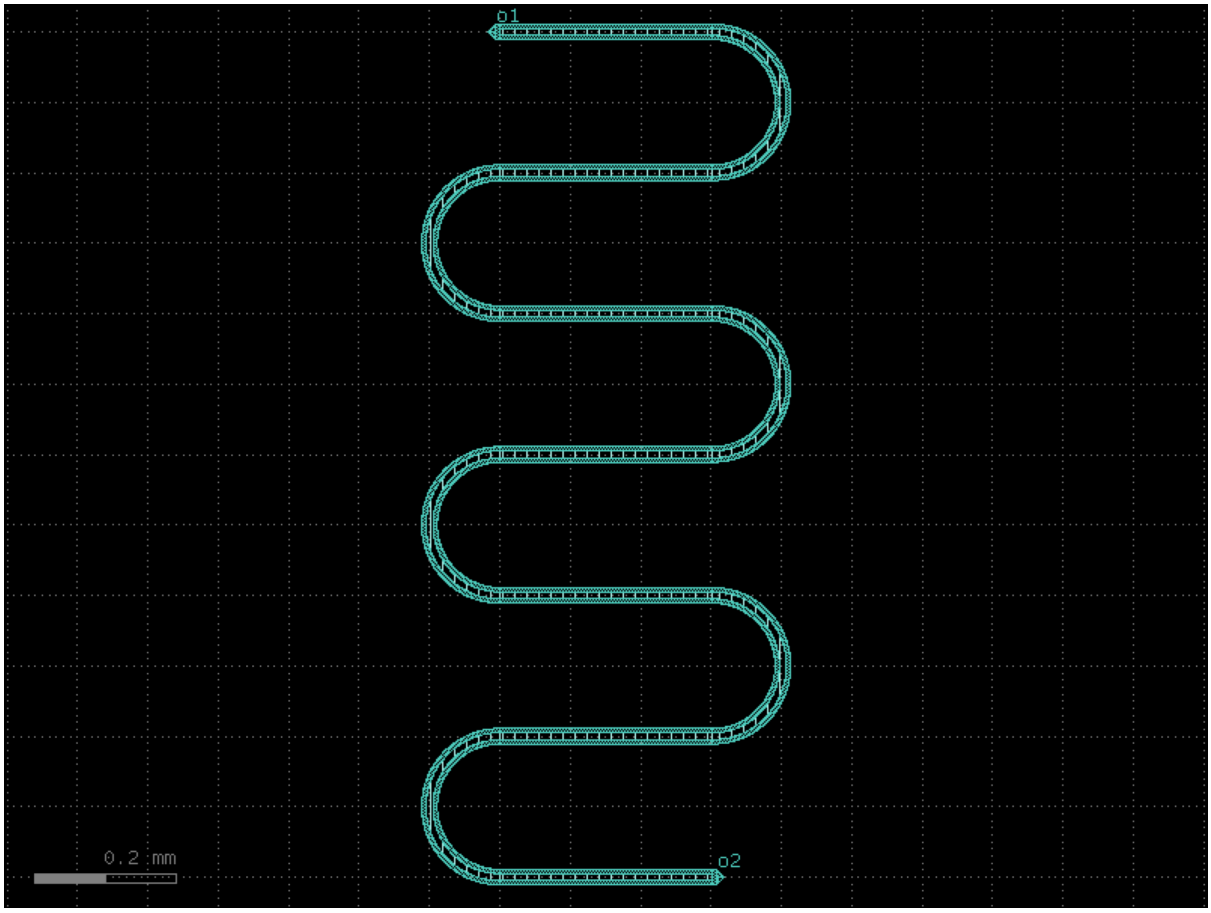
- **bend_spec** (*ComponentSpec*) – Specification for the bend component used in meanders.
- **cross_section** (*CrossSectionSpec*) – Cross-section specification for the resonator.
- **open_start** (*bool*) – If True, adds an etch section at the start of the resonator.
- **open_end** (*bool*) – If True, adds an etch section at the end of the resonator.

Returns
A gdsfactory component with meandering resonator geometry.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.resonator_half_wave(length=4000, meanders=6, cross_section='cpw', open_
↪start=True, open_end=True).copy()
c.draw_ports()
c.plot()
```



1.1.24 resonator_quarter_wave

qpdk.cells.**resonator_quarter_wave**(*length=4000.0*, *meanders=6*, *bend_spec=<function bend_circular>*, *cross_section='cpw'*, ***, *open_start=False*, *open_end=True*)

Creates a meandering coplanar waveguide resonator.

Changing *open_start* and *open_end* appropriately allows creating a shorted quarter-wave resonator or an open half-wave resonator.

See [MP12] for details

Parameters

- **length** (*float*) – Length of the resonator in µm.
- **meanders** (*int*) – Number of meander sections to fit the resonator in a compact area.
- **bend_spec** (*ComponentSpec*) – Specification for the bend component used in meanders.
- **cross_section** (*CrossSectionSpec*) – Cross-section specification for the resonator.
- **open_start** (*bool*) – If True, adds an etch section at the start of the resonator.
- **open_end** (*bool*) – If True, adds an etch section at the end of the resonator.

Returns
A gdsfactory component with meandering resonator geometry.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.resonator_quarter_wave(length=4000, meanders=6, cross_section='cpw',␣
↪open_start=False, open_end=True).copy()
c.draw_ports()
c.plot()
```

1.1.25 ring

qpdk.cells.**ring**(*radius=10.0*, *width=0.5*, *angle_resolution=2.5*, *layer='WG'*, *angle=360*)

Returns a ring.

Parameters

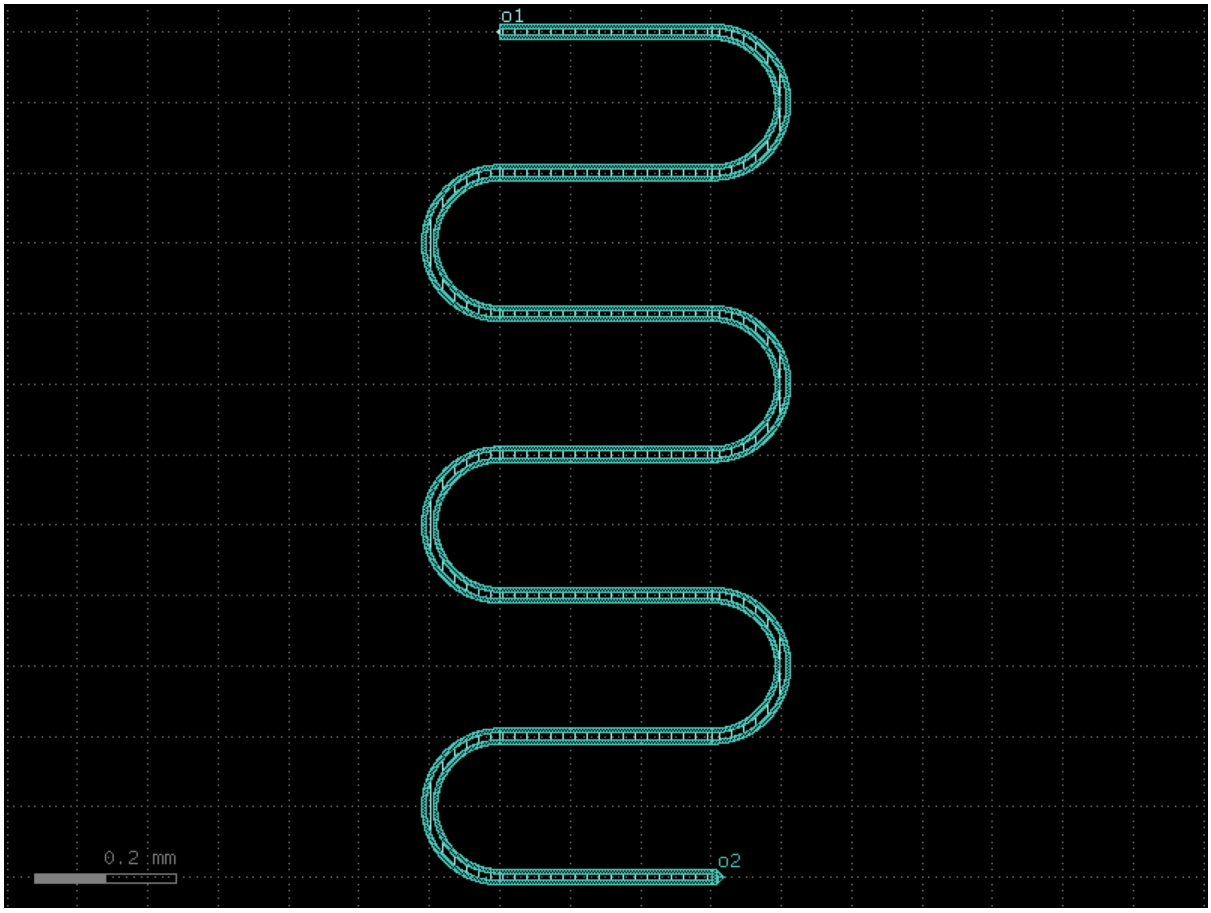
- **radius** (*float*) – ring radius.
- **width** (*float*) – of the ring.
- **angle_resolution** (*float*) – number of points per degree.
- **layer** (*LayerSpec*) – layer.
- **angle** (*float*) – angular coverage of the ring

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
```

(continues on next page)



(continued from previous page)

```
c = cells.ring(radius=10, width=0.5, angle_resolution=2.5, layer='WG', angle=360).
↪copy()
c.draw_ports()
c.plot()
```

1.1.26 single_josephson_junction_wire

qpdk.cells.**single_josephson_junction_wire**(***kwargs*)

Creates a single wire to use in a Josephson junction.

Parameters

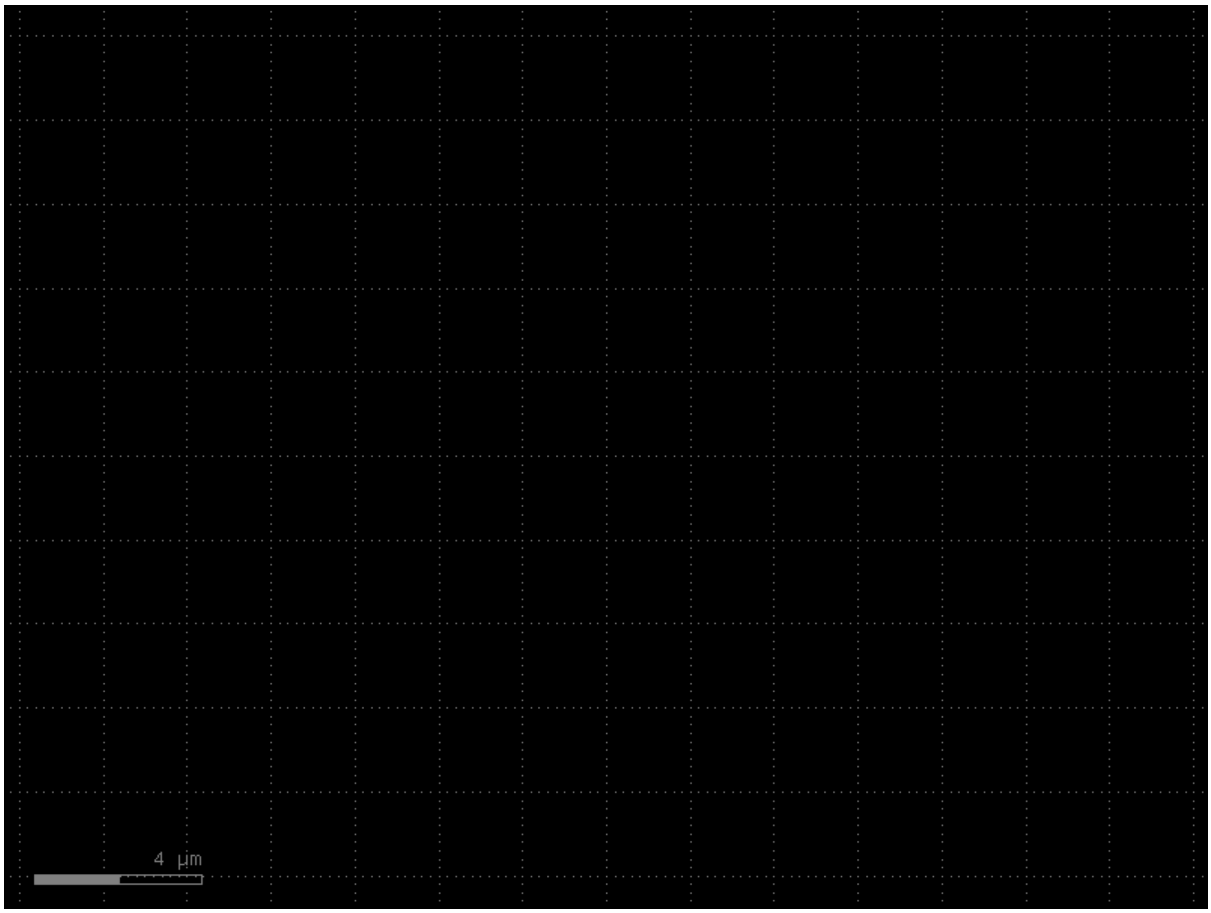
kwargs (*Unpack[SingleJosephsonJunctionWireParams]*) – SingleJosephsonJunctionWireParams parameters.

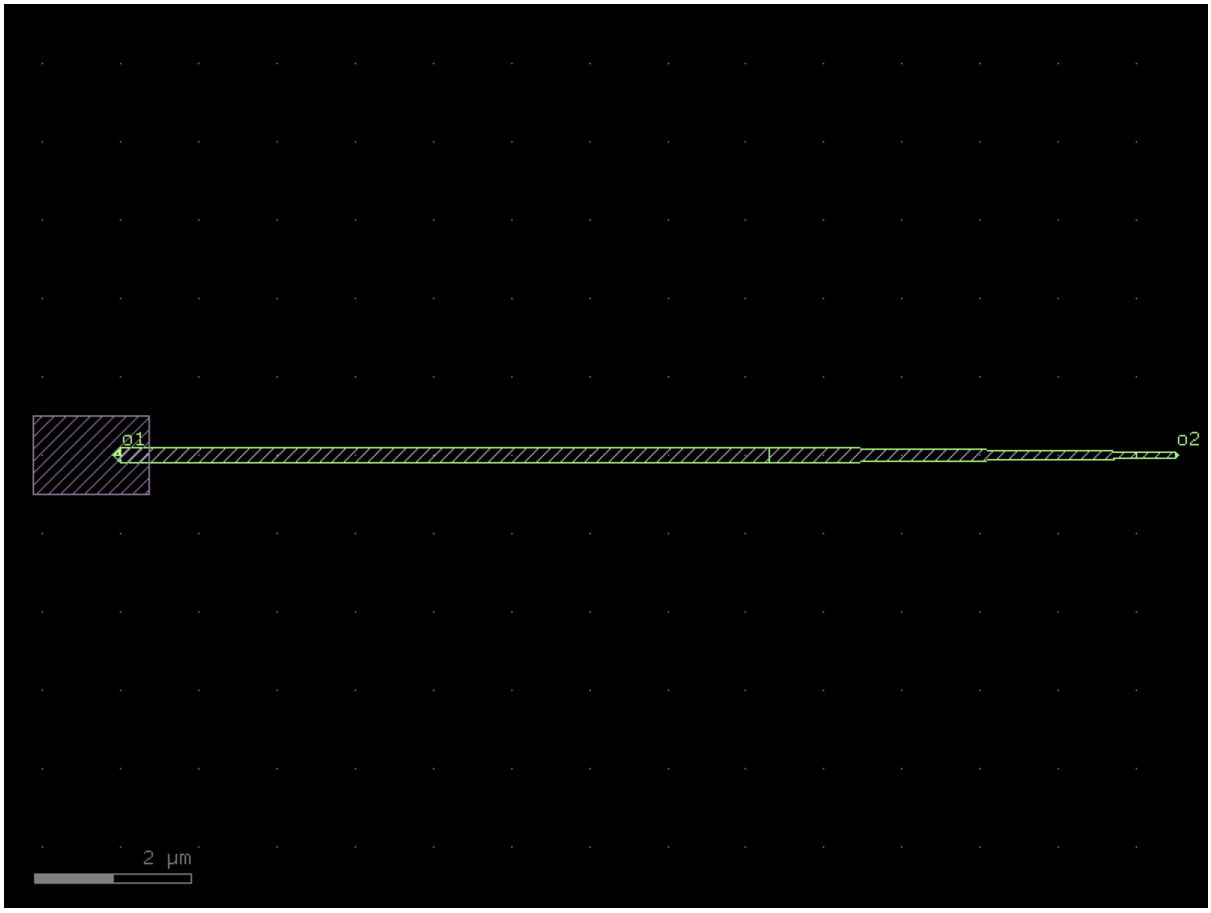
Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.single_josephson_junction_wire().copy()
c.draw_ports()
c.plot()
```


4 µm


o1
o2
2 µm

1.1.27 snspd

qpdk.cells.**snspd**(*wire_width=0.2*, *wire_pitch=0.6*, *size=(10*, *8)*, *num_squares=None*, *turn_ratio=4*, *terminals_same_side=False*, ***, *layer=<LayerMapQPDK.M1_DRAW: 1>*, *port_type='electrical'*)

Creates an optimally-rounded SNSPD.

Parameters

- **wire_width** (*float*) – Width of the wire.
- **wire_pitch** (*float*) – Distance between two adjacent wires. Must be greater than *width*.
- **size** (*Size*) – Float2 (width, height) of the rectangle formed by the outer boundary of the SNSPD.
- **num_squares** (*int | None*) – int | None = None Total number of squares inside the SNSPD length.
- **turn_ratio** (*float*) – float Specifies how much of the SNSPD width is dedicated to the 180 degree turn. A *turn_ratio* of 10 will result in 20% of the width being comprised of the turn.
- **terminals_same_side** (*bool*) – If True, both ports will be located on the same side of the SNSPD.
- **layer** (*LayerSpec*) – layer spec to put polygon geometry on.
- **port_type** (*str*) – type of port to add to the component.

Return type
Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.snspd(wire_width=0.2, wire_pitch=0.6, size=(10, 8), turn_ratio=4,␣
↪terminals_same_side=False, layer='M1_DRAW', port_type='electrical').copy()
c.draw_ports()
c.plot()
```

1.1.28 squid_junction

qpdk.cells.**squid_junction**(*junction_spec=<function josephson_junction>*, *loop_area=4*)

Creates a SQUID (Superconducting Quantum Interference Device) junction component.

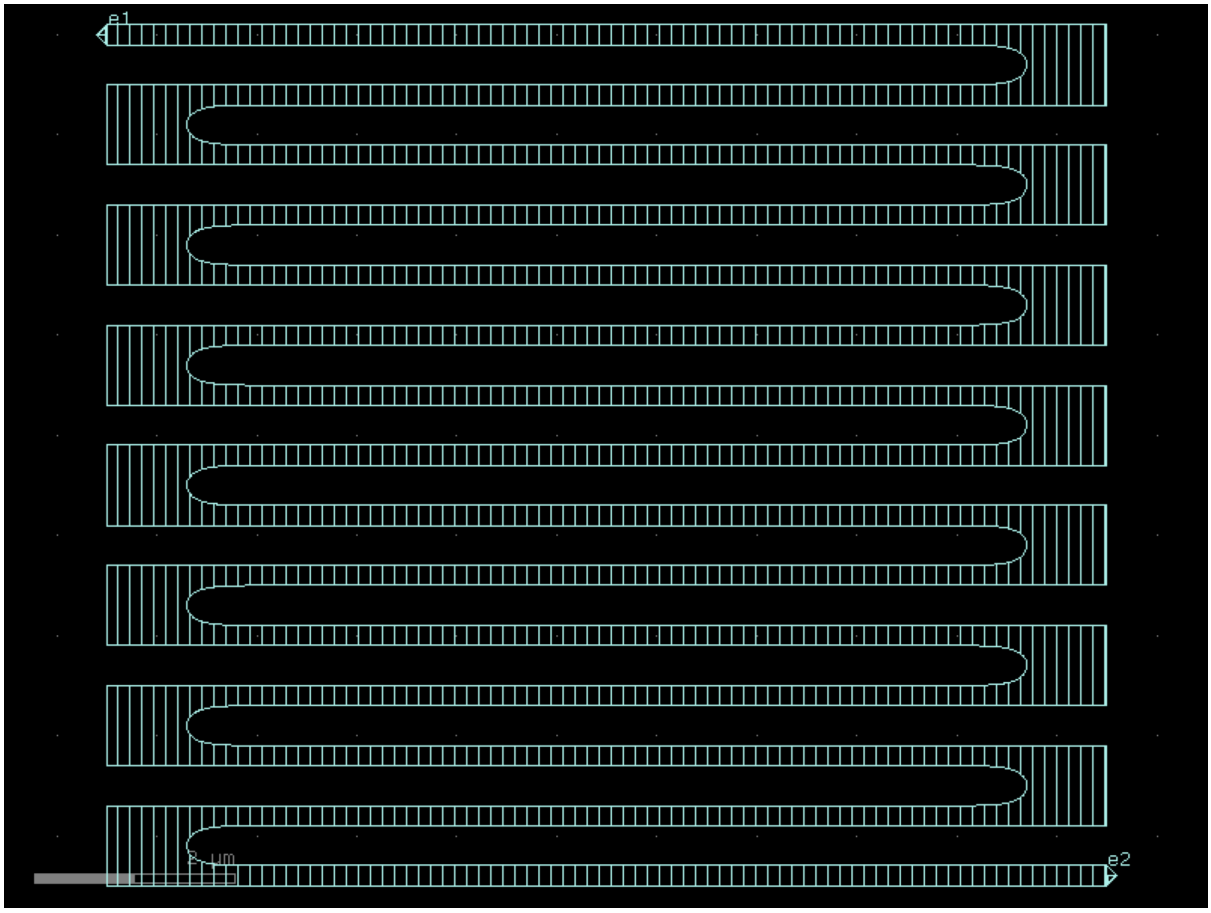
A SQUID consists of two Josephson junctions connected in parallel, forming a loop.

See [CB04] for details.

Parameters

- **junction_spec** (*ComponentSpec*) – Component specification for the Josephson junction component.
- **loop_area** (*float*) – Area of the SQUID loop in μm^2. This does not take into account the junction wire widths.

Return type
Component




```
from qpdk import cells, PDK

PDK.activate()
c = cells.squid_junction(loop_area=4).copy()
c.draw_ports()
c.plot()
```

1.1.29 straight

qpdk.cells.**straight**(***kwargs*)

Returns a Straight waveguide.

Parameters

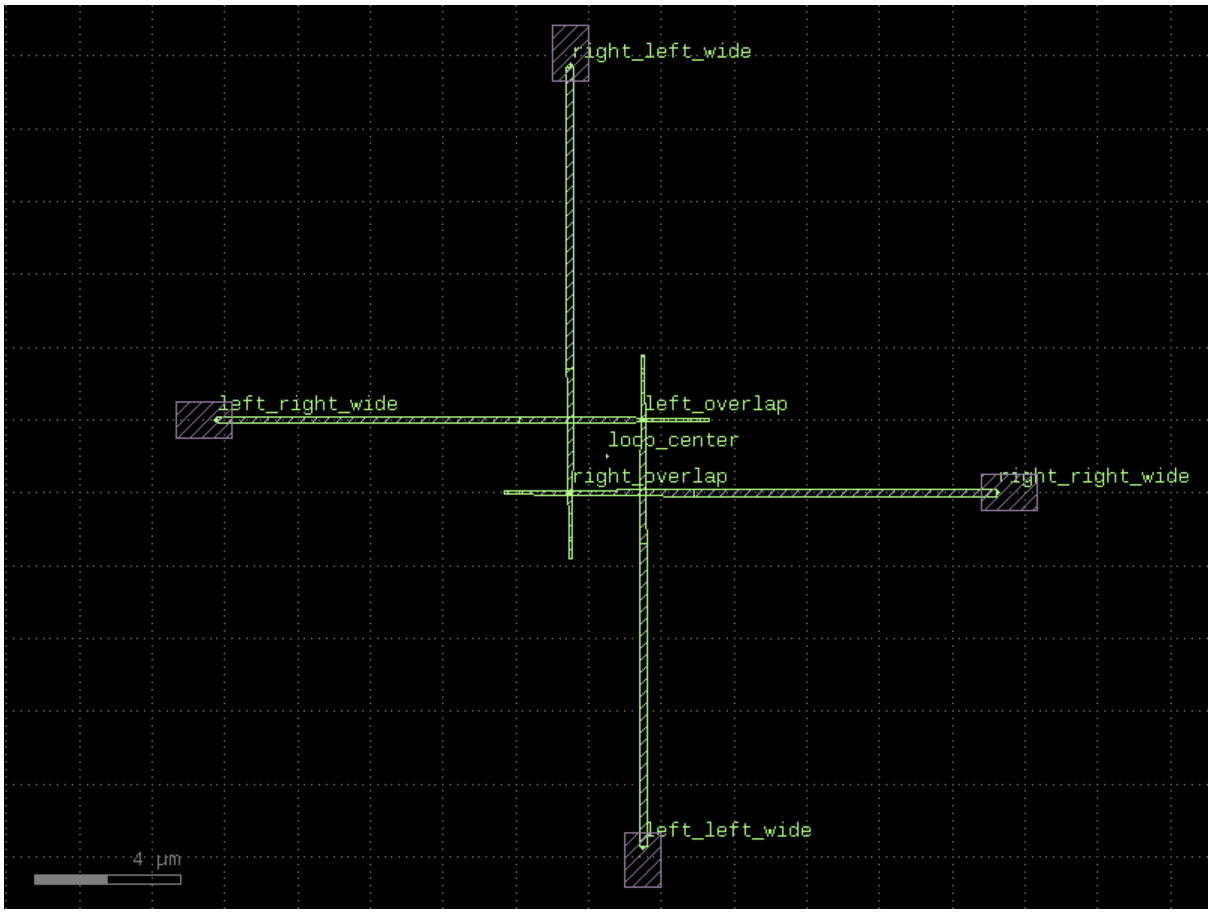
****kwargs** (*Unpack[StraightKwargs]*) – Arguments passed to gf.c.straight.

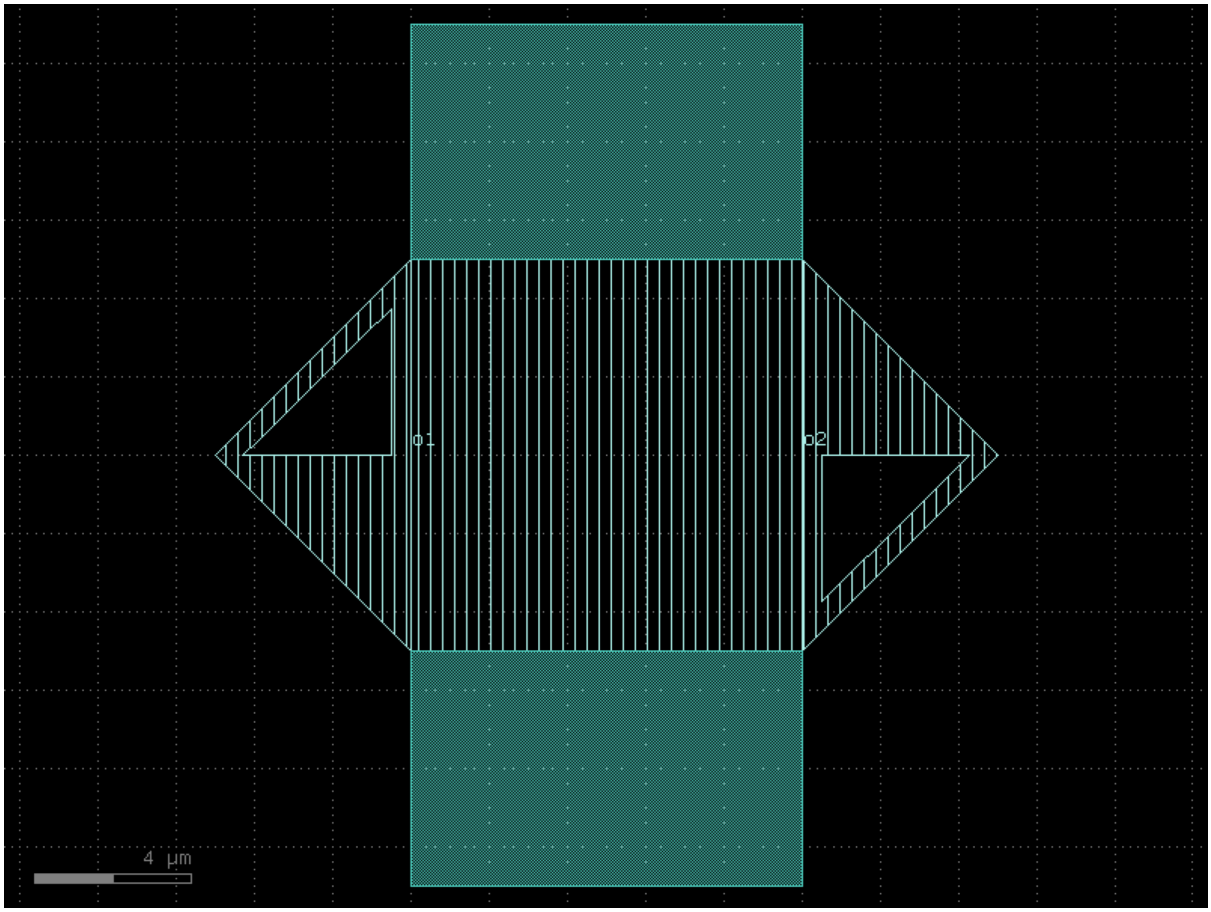
Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.straight().copy()
c.draw_ports()
c.plot()
```


right_left_wide
left_right_wide
left_overlap
loop_center
right_overlap
right_right_wide
left_left_wide
4 µm


φ1
φ2
4 µm

1.1.30 straight_all_angle

qpdk.cells.**straight_all_angle**(***kwargs*)

Returns a Straight waveguide with offgrid ports.

Parameters

****kwargs** (*Unpack[StraightAllAngleKwargs]*) – Arguments passed to gf.c.straight_all_angle.

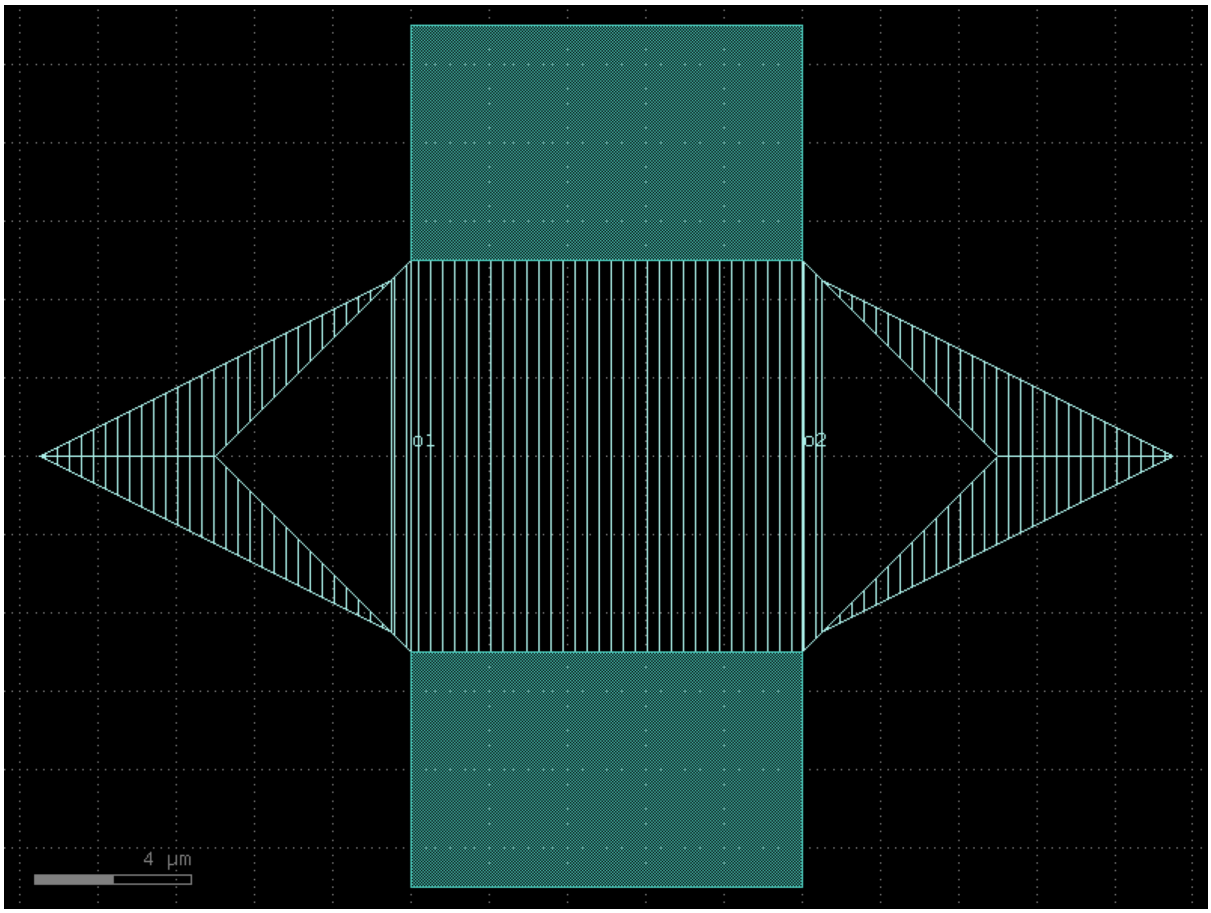
Return type

ComponentAllAngle

```
o1  ──────────────── o2
        length
```

```
from qpdk import cells, PDK

PDK.activate()
c = cells.straight_all_angle().copy()
c.draw_ports()
c.plot()
```



1.1.31 taper_cross_section

qpdk.cells.**taper_cross_section**(*, *cross_section1='cpw'*, *cross_section2='cpw'*, *length=10*, *npoints=100*, *linear=False*, *width_type='sine'*)

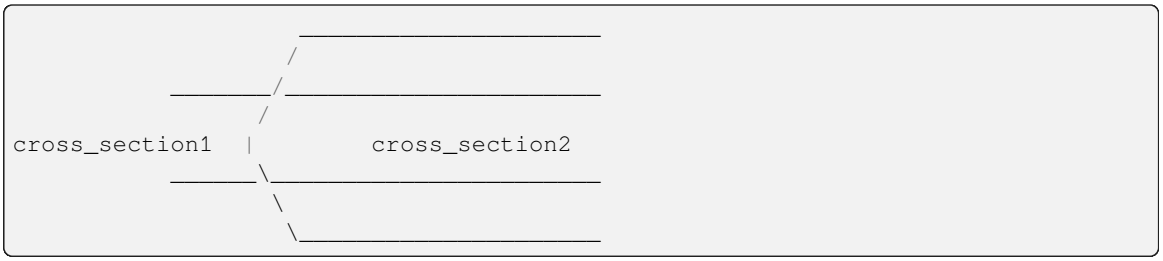
Returns taper transition between cross_section1 and cross_section2.

Parameters

- **cross_section1** (*CrossSectionSpec*) – start cross_section factory.
- **cross_section2** (*CrossSectionSpec*) – end cross_section factory.
- **length** (*float*) – transition length.
- **npoints** (*int*) – number of points.
- **linear** (*bool*) – shape of the transition, sine when False.
- **width_type** (*str*) – shape of the transition ONLY IF linear is False

Return type

Component




```
from qpdk import cells, PDK

PDK.activate()
c = cells.taper_cross_section(cross_section1='cpw', cross_section2='cpw',␣
↪length=10, npoints=100, linear=False, width_type='sine').copy()
c.draw_ports()
c.plot()
```

1.1.32 tee

qpdk.cells.**tee**(*cross_section='cpw'*)

Returns a three-way tee waveguide.

Parameters

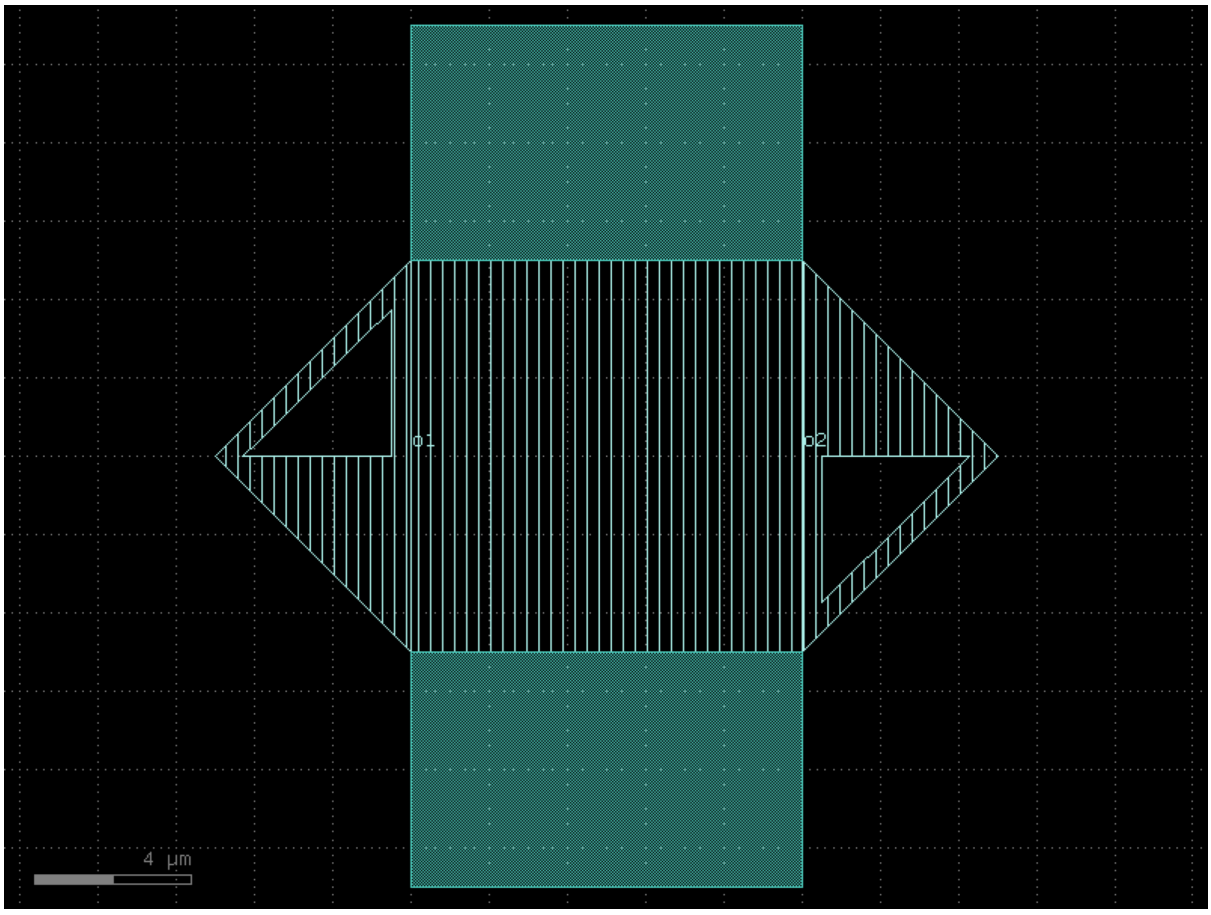
cross_section(*CrossSection | str | dict[str, Any] | Callable[[...], CrossSection] | SymmetricalCrossSection | DCrossSection*) – specification (CrossSection, string or dict).

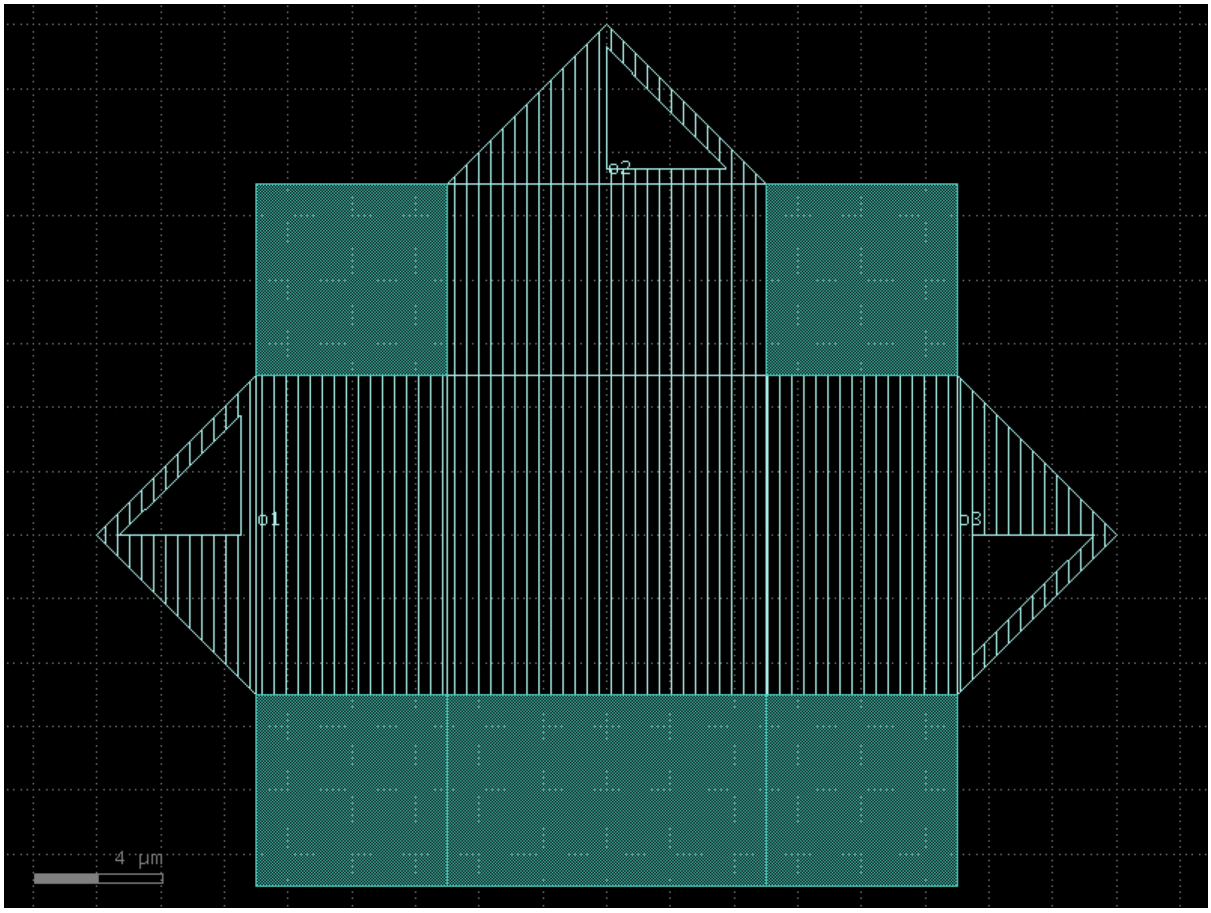
Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.tee(cross_section='cpw').copy()
c.draw_ports()
c.plot()
```


o1
o2
4 µm


o2
o1
o3
4 µm

1.1.33 transform_component

qpdk.cells.**transform_component**(*component*, *transform*)

Applies a complex transformation to a component.

For use with `container()`.

Parameters

- **component** (*Component*)
- **transform** (*DCplxTrans*)

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.transform_component().copy()
c.draw_ports()
c.plot()
```

1.1.34 transmon_with_resonator

qpdk.cells.**transmon_with_resonator**(*transmon='double_pad_transmon_with_bbox'*, *resonator=functools.partial(<function resonator>, open_start=False, open_end=True, length=4000, meanders=6)*, *resonator_meander_start=(-700, -1300)*, *resonator_length=5000.0*, *resonator_params=None*, *coupler=functools.partial(<function plate_capacitor_single>, thickness=20, fingers=18)*)

Returns a transmon qubit coupled to a quarter wave resonator.

Parameters

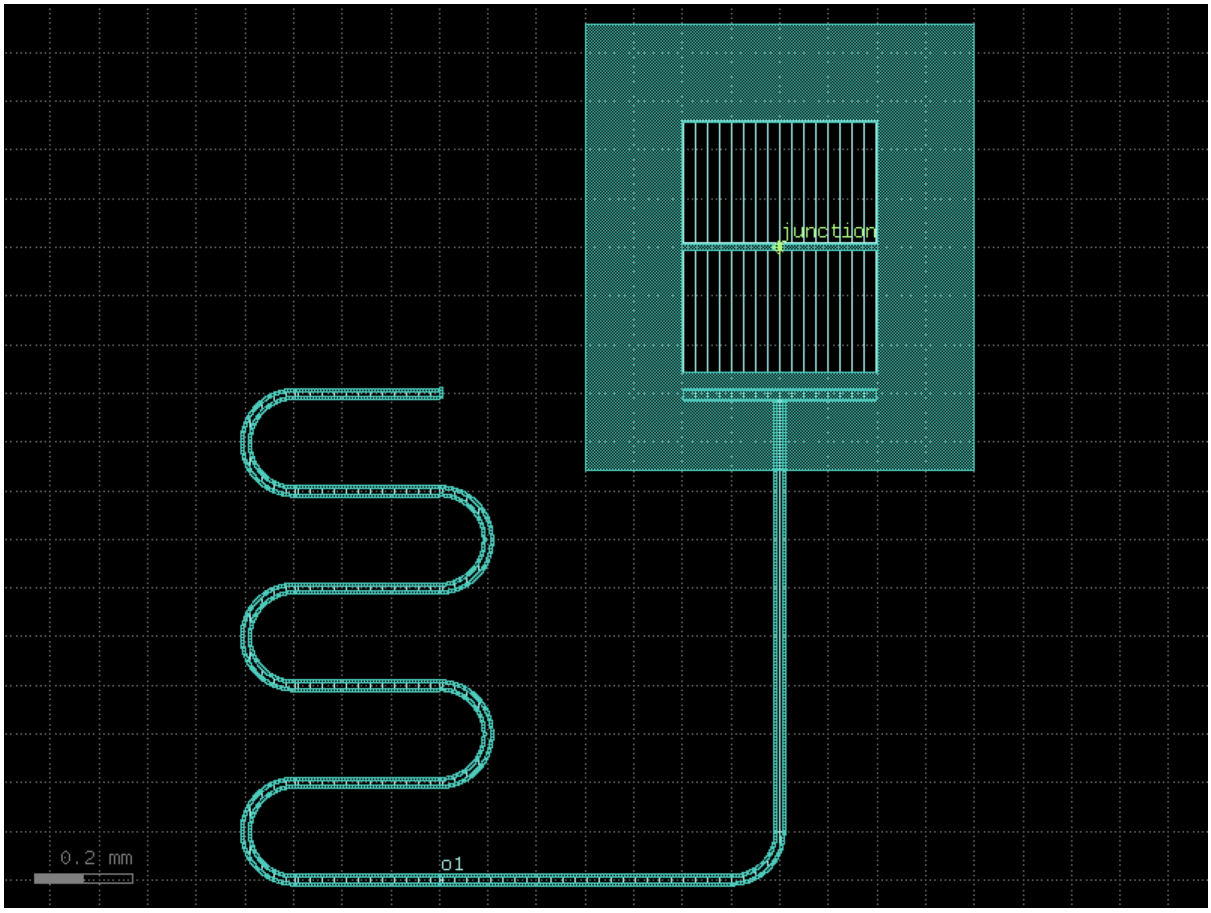
- **transmon** (*ComponentSpec*) – Transmon component.
- **resonator** (*ComponentSpec*) – Resonator component.
- **resonator_meander_start** (*tuple[float, float]*) – (x, y) position of the start of the resonator meander.
- **resonator_length** (*float*) – Length of the resonator in µm.
- **resonator_params** (*ResonatorParams | None*) – Parameters for the resonator component if it accepts any.
- **coupler** (*ComponentSpec*) – Coupler component.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.transmon_with_resonator(transmon='double_pad_transmon_with_bbox',␣
 ↪resonator_meander_start=(-700, -1300), resonator_length=5000).copy()
c.draw_ports()
c.plot()
```



1.1.35 tsv

qpdk.cells.**tsv**(*diameter=15.0*)

Creates a Through-silicon via (TSV) component for 3D integration.

See [YSM+20].

Parameters

diameter (*float*) – Diameter of the via in µm.

Returns

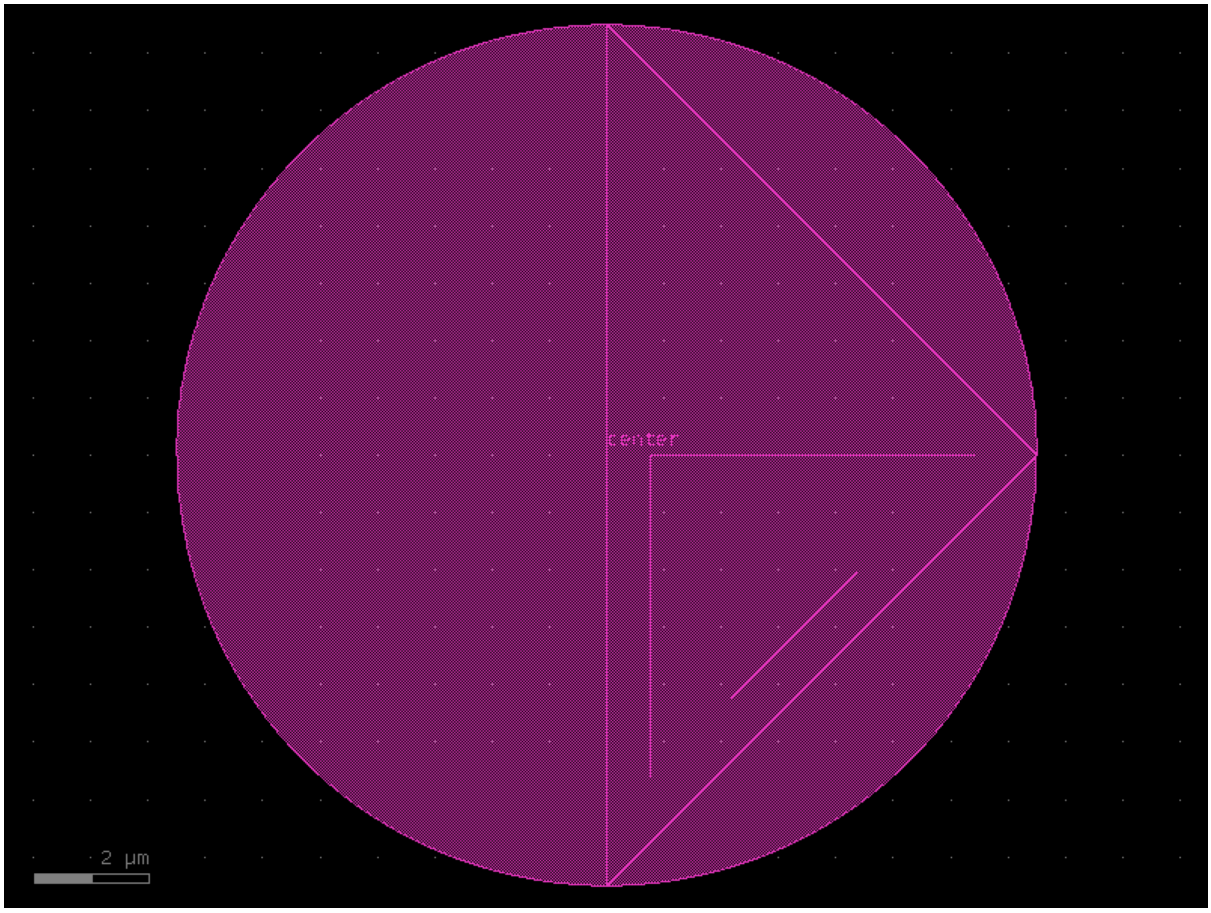
A gdsfactory Component representing the TSV.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.tsv(diameter=15).copy()
c.draw_ports()
c.plot()
```



1.1.36 xmon_transmon

qpdk.cells.**xmon_transmon**(***kwargs*)

Creates an Xmon style transmon qubit with cross-shaped geometry.

An Xmon transmon consists of a cross-shaped capacitor pad with four arms extending from a central region, connected by a Josephson junction at the center. The design provides better control over the coupling to readout resonators and neighboring qubits through the individual arm geometries.

See [BKM+13] for details about the Xmon design.

Parameters

****kwargs** (*Unpack[XmonTransmonParams]*) – XmonTransmonParams for the Xmon transmon qubit.

Returns

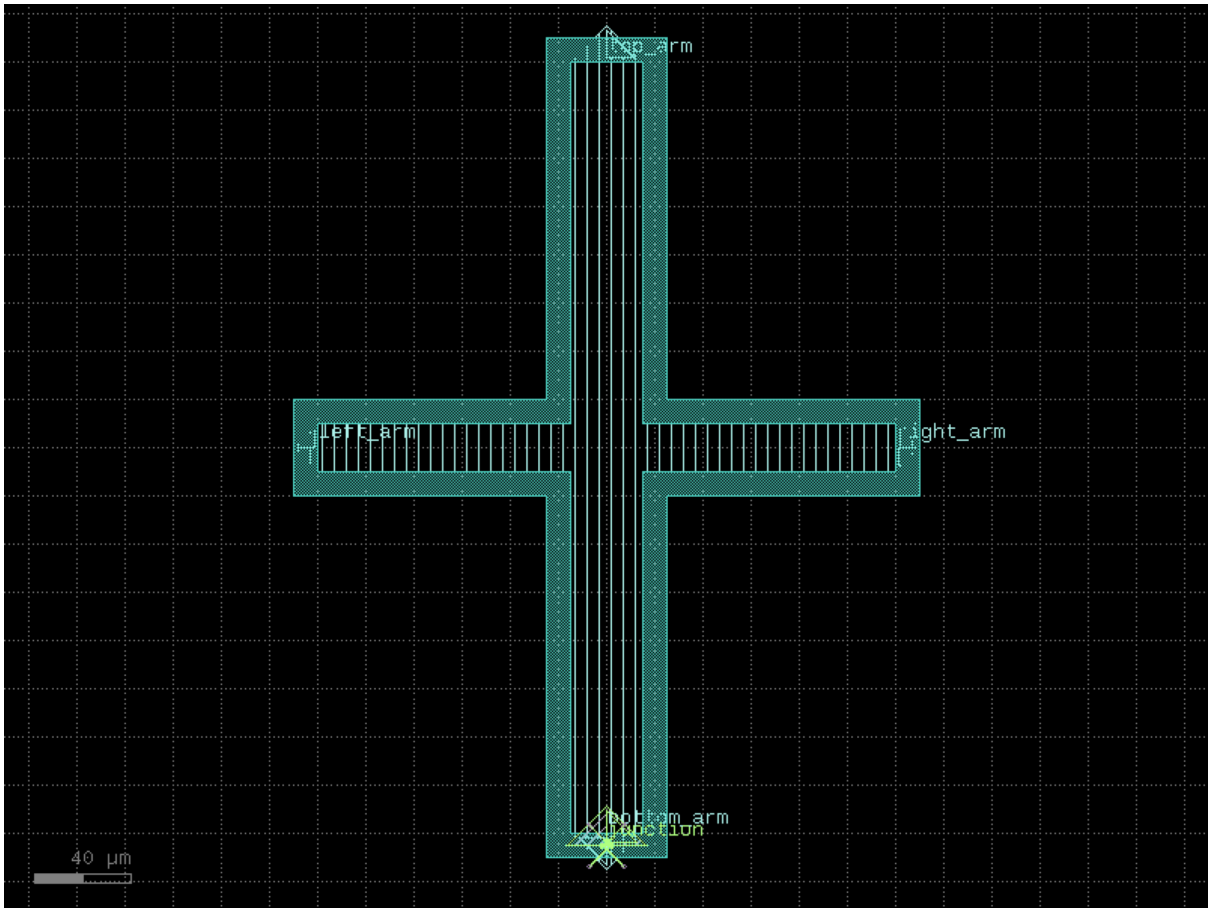
A gdsfactory component with the Xmon transmon geometry.

Return type

Component

```
from qpdk import cells, PDK

PDK.activate()
c = cells.xmon_transmon().copy()
c.draw_ports()
c.plot()
```



1.2 References

1.3 Models

Model definitions for qpdk.

`qpdk.models.`**`resonator_frequency`**(*length*, *media*, *is_quarter_wave=True*)

Calculate the resonance frequency of a quarter-wave resonator.

$$f = \frac{v_p}{4L} \text{ (quarter-wave resonator)}$$
$$f = \frac{v_p}{2L} \text{ (half-wave resonator)}$$

There is some variation according to the frequency range specified for `media` due to how v_p is calculated in skrf. The phase velocity is given by $v_p = i \cdot \omega/\gamma$, where γ is the complex propagation constant and ω is the angular frequency.

See [MP12, Sim01] for details.

Parameters

- **`length`** (*`float`*) – Length of the resonator in µm.
- **`media`** (*`Media`*) – skrf media object defining the CPW (or other) properties.
- **`is_quarter_wave`** (*`bool`*) – If True, calculates for a quarter-wave resonator; if False, for a half-wave resonator. default is True.

Returns

Resonance frequency in Hz.

Return type
float

1.3.1 References

1.4 CHANGELOG

1.4.1 0.0.0

- first draft

Part II

Samples

2 Samples

2.1 qpdk.samples.filled_resonator.filled_quarter_wave_resonator

qpdk.samples.filled_resonator.**filled_quarter_wave_resonator**()

Returns a quarter wave resonator filled with magnetic vortex trapping rectangles.

This sample demonstrates how to use the fill_magnetic_vortices helper function to add small rectangles that trap magnetic vortices in superconducting quantum circuits.

Returns
A quarter wave resonator with fill rectangles for vortex trapping.

Return type
Component

```
import qpdk.samples.filled_resonator
from qpdk import PDK

PDK.activate()
c = qpdk.samples.filled_resonator.filled_quarter_wave_resonator().copy()
c.draw_ports()
c.plot()
```

2.2 qpdk.samples.filled_test_chip.filled_qubit_test_chip

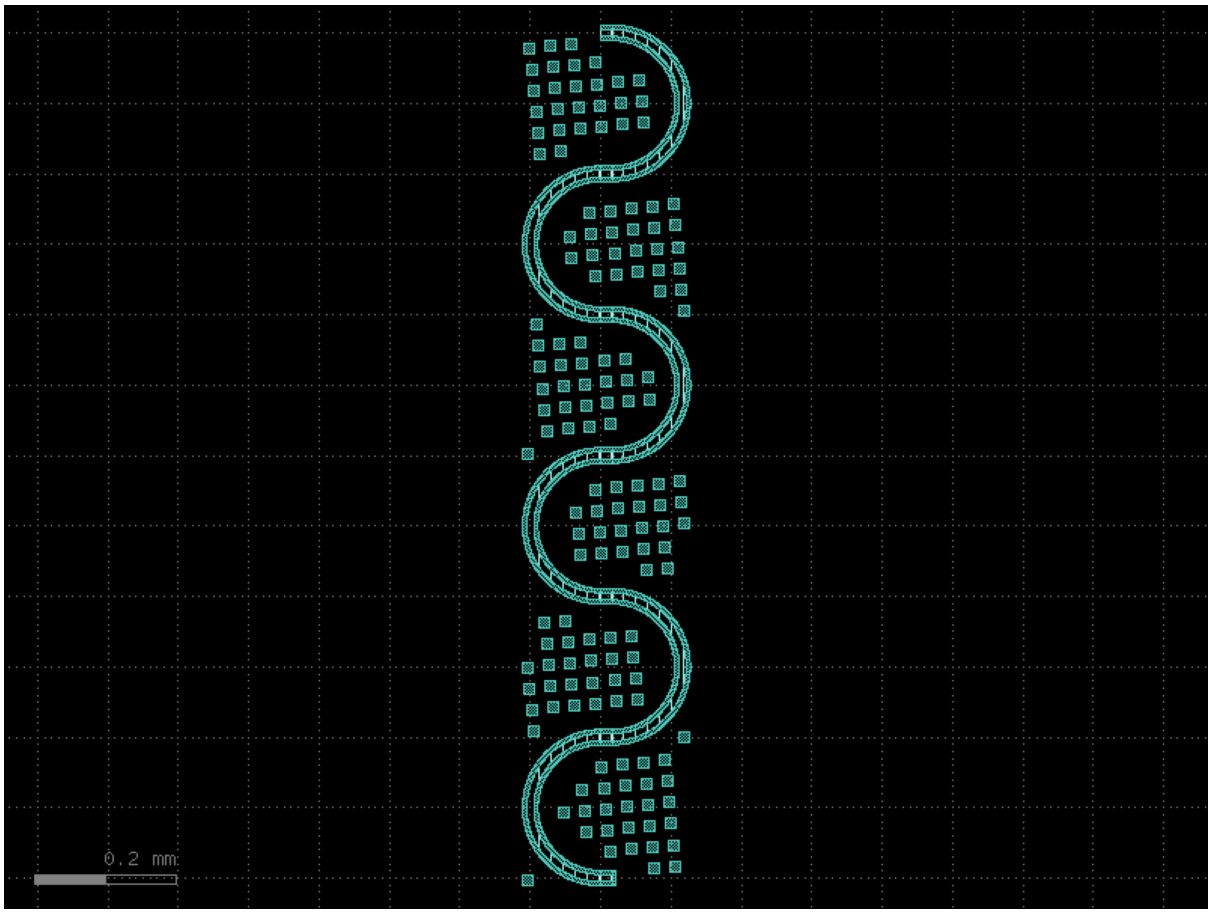
qpdk.samples.filled_test_chip.**filled_qubit_test_chip**()

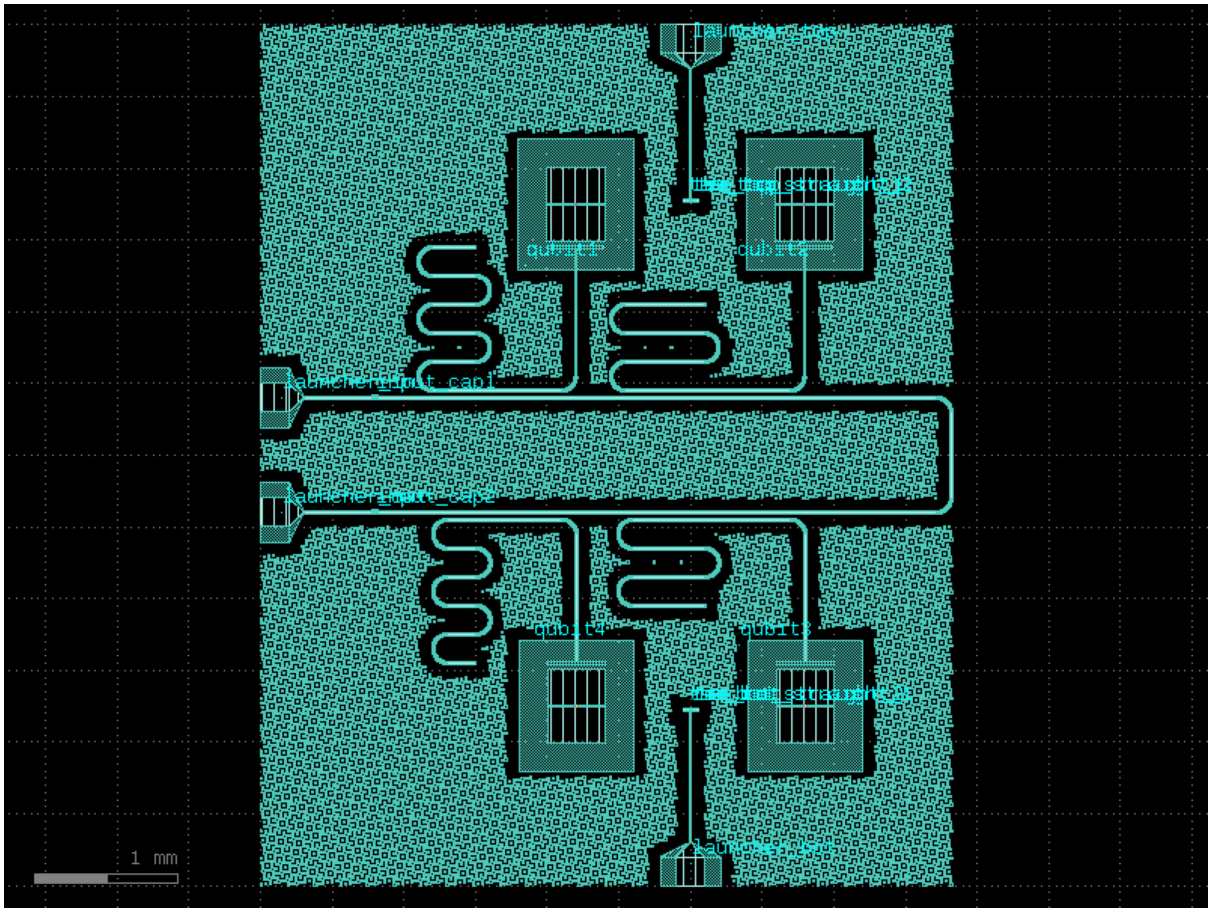
Returns a qubit test chip filled with magnetic vortex trapping rectangles.

Rouhly corresponds to the sample in [TSKivijarvi+25].

```
import qpdk.samples.filled_test_chip
from qpdk import PDK

PDK.activate()
c = qpdk.samples.filled_test_chip.filled_qubit_test_chip().copy()
c.draw_ports()
c.plot()
```


0.2 mm


qubit1
qubit4
qubit4
qubit3
1 mm

2.3 qpdk.samples.resonator_test_chip.filled_resonator_test_chip

qpdk.samples.resonator_test_chip.**filled_resonator_test_chip**()

Creates a resonator test chip filled with magnetic vortex trapping holes.

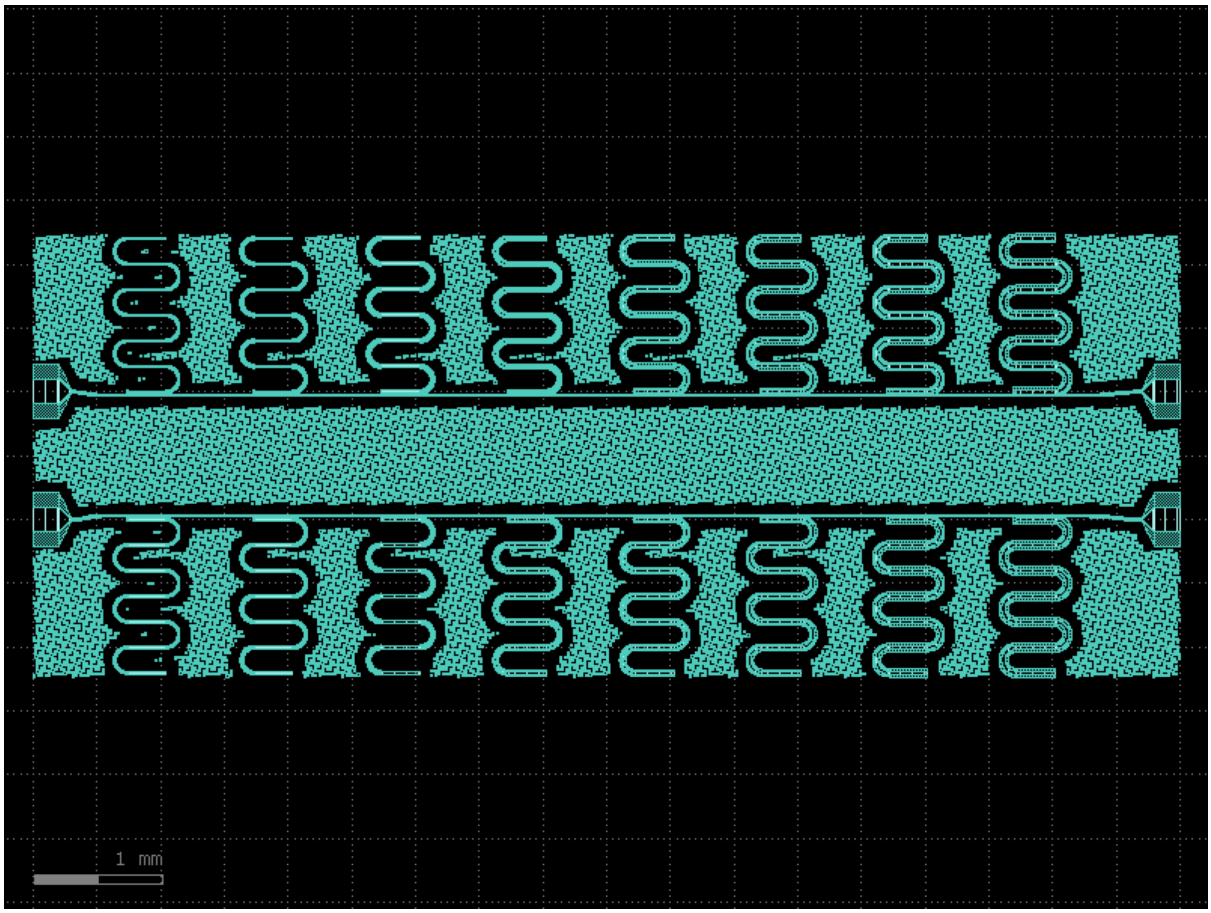
This version includes the complete resonator test chip layout with additional ground plane holes to trap magnetic vortices, improving the performance of superconducting quantum circuits.

Returns
Test chip with ground plane fill patterns.

Return type
Component

```
import qpdk.samples.resonator_test_chip
from qpdk import PDK

PDK.activate()
c = qpdk.samples.resonator_test_chip.filled_resonator_test_chip().copy()
c.draw_ports()
c.plot()
```



2.4 qpdk.samples.resonator_test_chip.resonator_test_chip

qpdk.samples.resonator_test_chip.**resonator_test_chip**(*probeline_length=9000.0*, *probeline_separation=1000.0*, *resonator_length=4000.0*, *coupling_length=200.0*, *coupling_gap=16.0*)

Creates a resonator test chip with two probelines and 16 resonators.

The chip features two horizontal probelines running west to east, each with launchers on both ends. Eight quarter-wave resonators are coupled to each probeline, with systematically varied cross-section parameters for characterization studies.

Parameters

- **probeline_length** (*float*) – Length of each probeline in µm.
- **probeline_separation** (*float*) – Vertical separation between probelines in µm.
- **resonator_length** (*float*) – Length of each resonator in µm.
- **coupling_length** (*float*) – Length of coupling region between resonator and probeline in µm.
- **coupling_gap** (*float*) – Gap between resonator and probeline for coupling in µm.

Returns
A gdsfactory component containing the complete test chip layout.

Return type
Component

```
import qpdk.samples.resonator_test_chip
from qpdk import PDK

PDK.activate()
c = qpdk.samples.resonator_test_chip.resonator_test_chip(probeline_length=9000,␣
↪probeline_separation=1000, resonator_length=4000, coupling_length=200, coupling_
↪gap=16).copy()
c.draw_ports()
c.plot()
```

2.5 qpdk.samples.sample0.sample0_hello_world

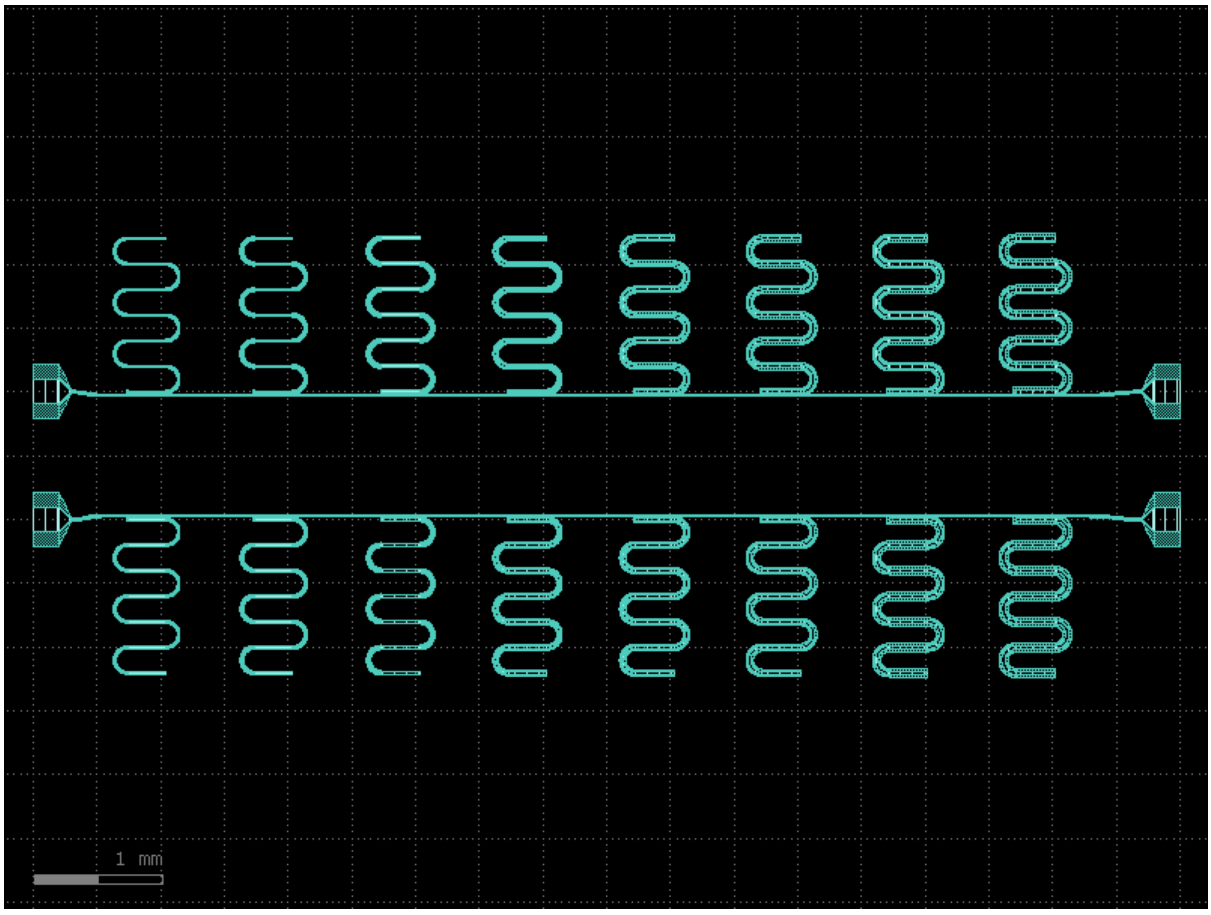
qpdk.samples.sample0.**sample0_hello_world**()

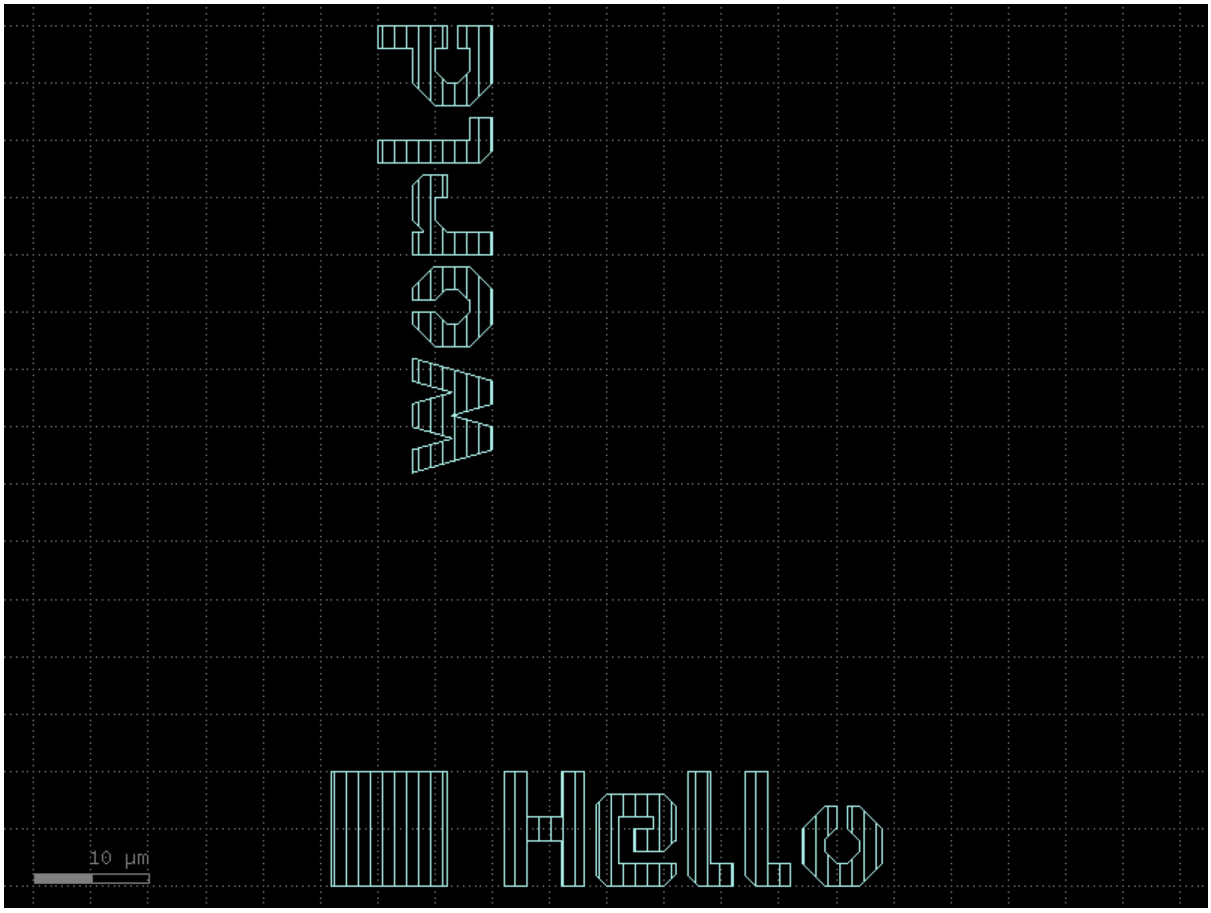
Returns a component with 'Hello world' text and a rectangle.

Return type
Component

```
import qpdk.samples.sample0
from qpdk import PDK

PDK.activate()
c = qpdk.samples.sample0.sample0_hello_world().copy()
c.draw_ports()
c.plot()
```


1 mm


10 µm

2.6 qpdk.samples.sample1.sample1_connect

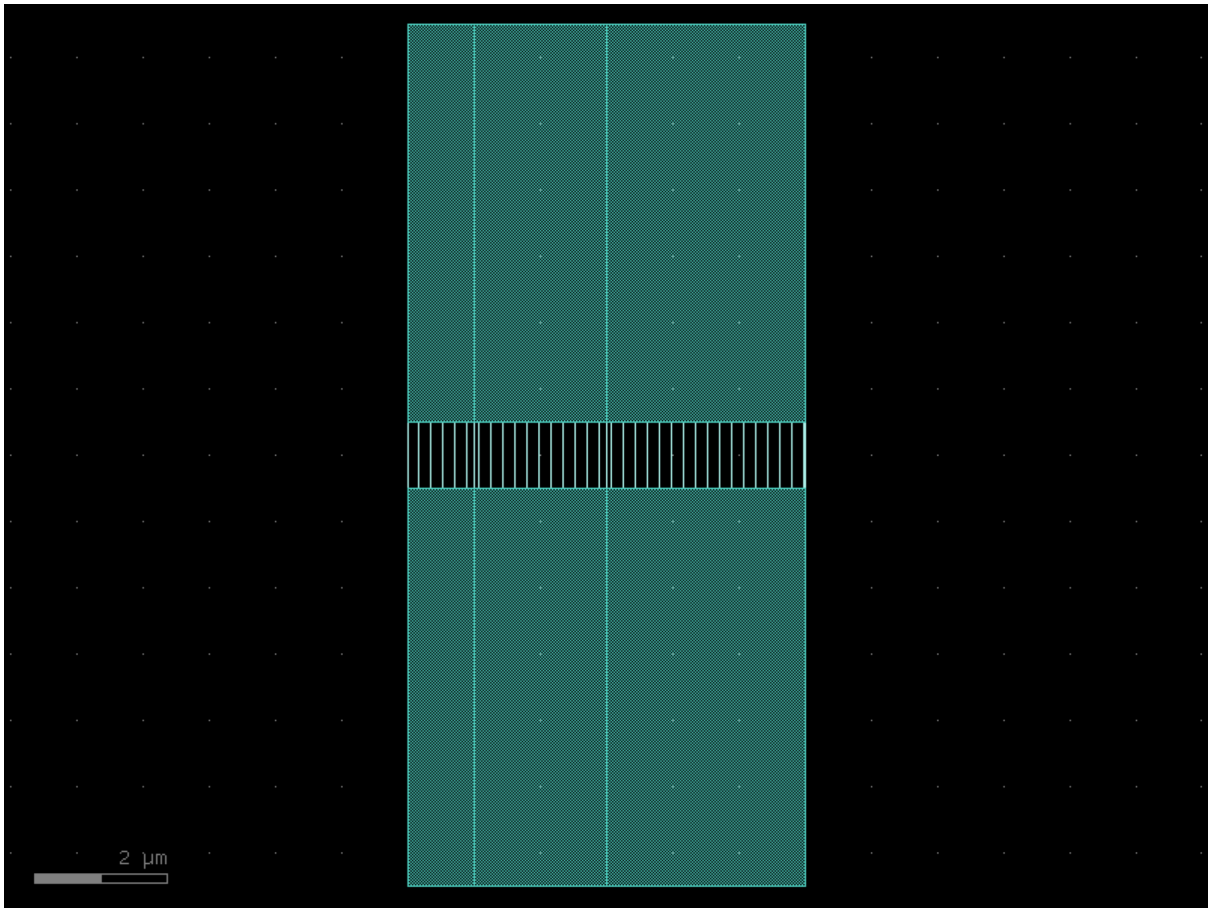
qpdk.samples.sample1.**sample1_connect**()

Returns a component with connected waveguides.

Return type
Component

```
import qpdk.samples.sample1
from qpdk import PDK

PDK.activate()
c = qpdk.samples.sample1.sample1_connect().copy()
c.draw_ports()
c.plot()
```



2.7 qpdk.samples.sample2.sample2_remove_layers

qpdk.samples.sample2.**sample2_remove_layers**()

Returns a component with 'Hello world' text and a rectangle.

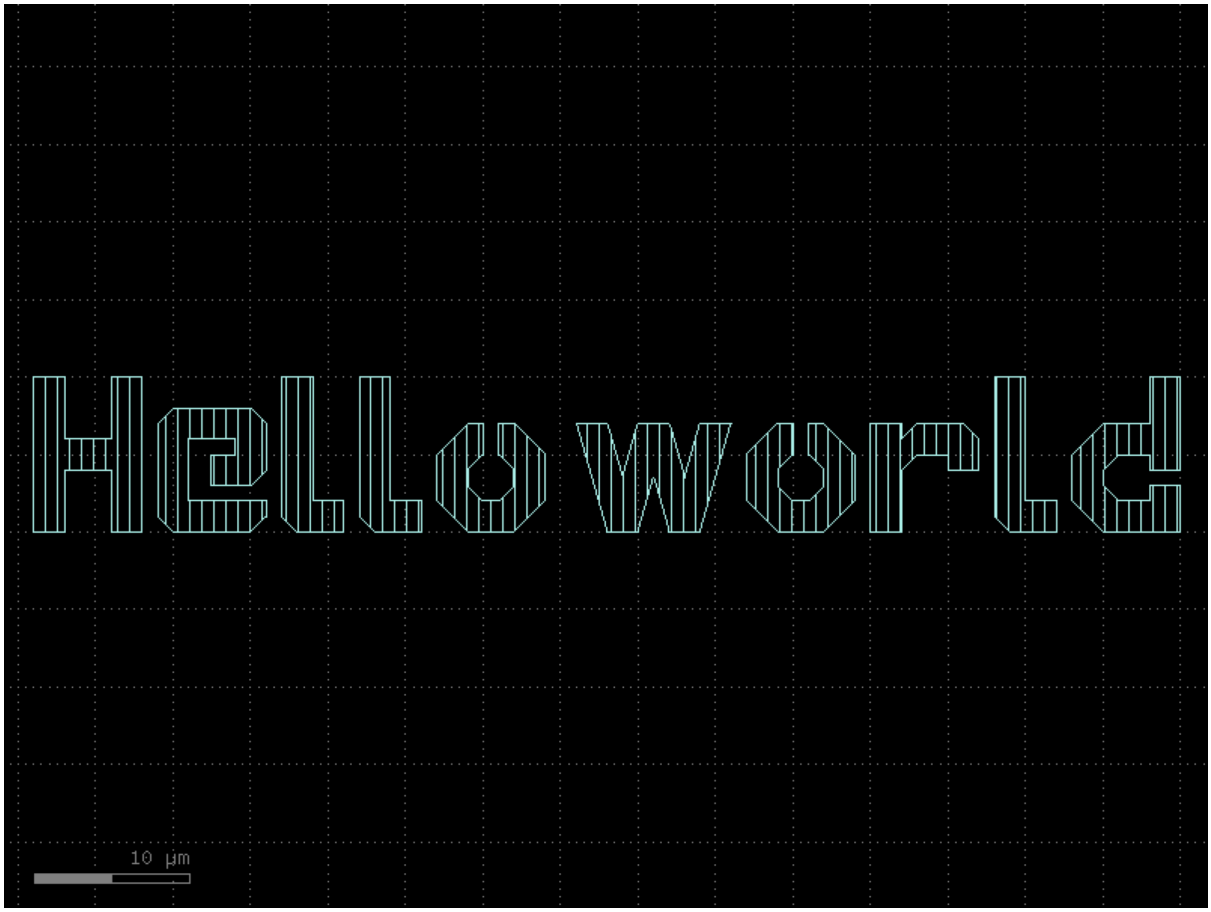
Return type
Component

```
import qpdk.samples.sample2
from qpdk import PDK
```

(continues on next page)

(continued from previous page)

```
PDK.activate()
c = qpdk.samples.sample2.sample2_remove_layers().copy()
c.draw_ports()
c.plot()
```



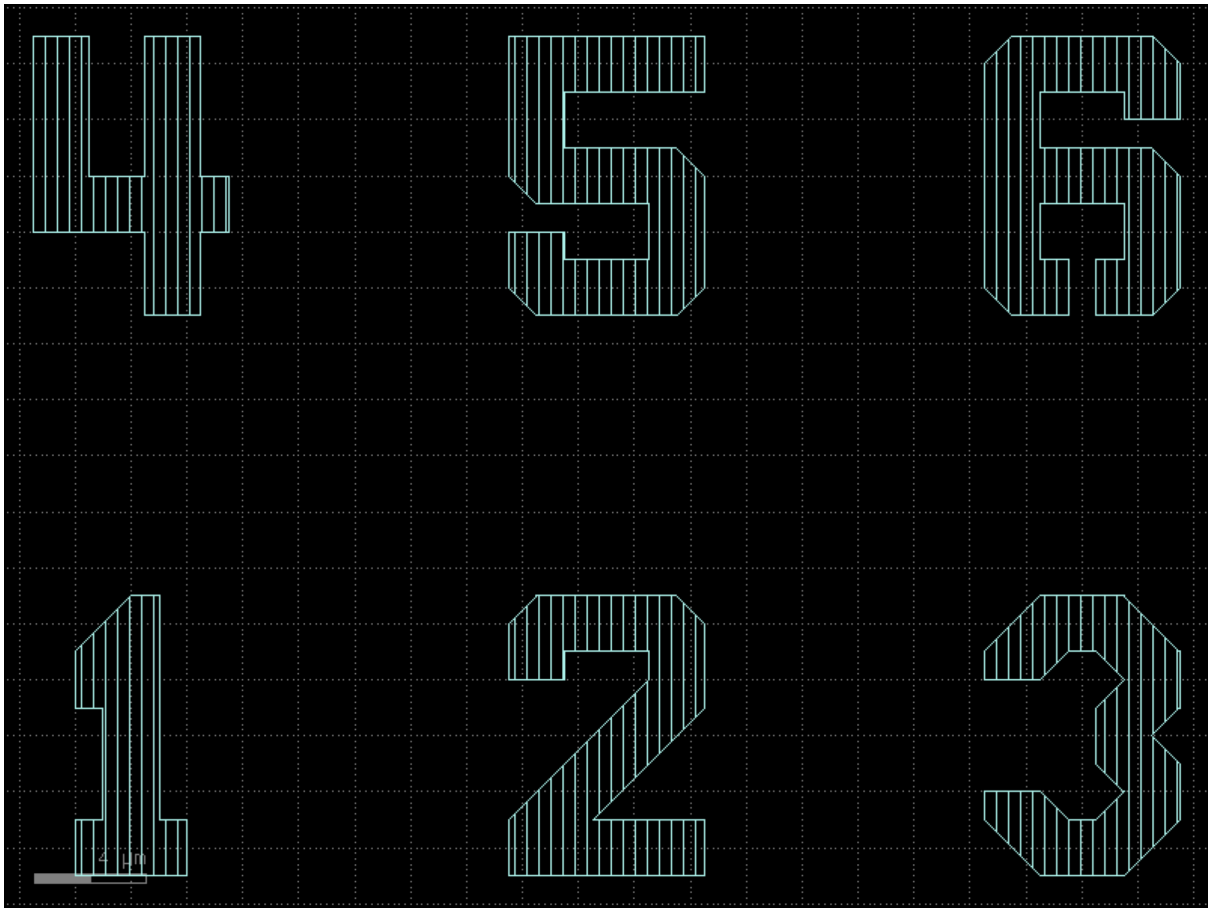
2.8 qpdk.samples.sample3.sample3_grid

qpdk.samples.sample3.**sample3_grid**()

Returns a component with a grid of text elements.

```
import qpdk.samples.sample3
from qpdk import PDK

PDK.activate()
c = qpdk.samples.sample3.sample3_grid().copy()
c.draw_ports()
c.plot()
```



2.9 qpdk.samples.sample4.sample4_pack

qpdk.samples.sample4.**sample4_pack**()

Returns a component with a packed set of ellipses.

```
import qpdk.samples.sample4
from qpdk import PDK

PDK.activate()
c = qpdk.samples.sample4.sample4_pack().copy()
c.draw_ports()
c.plot()
```

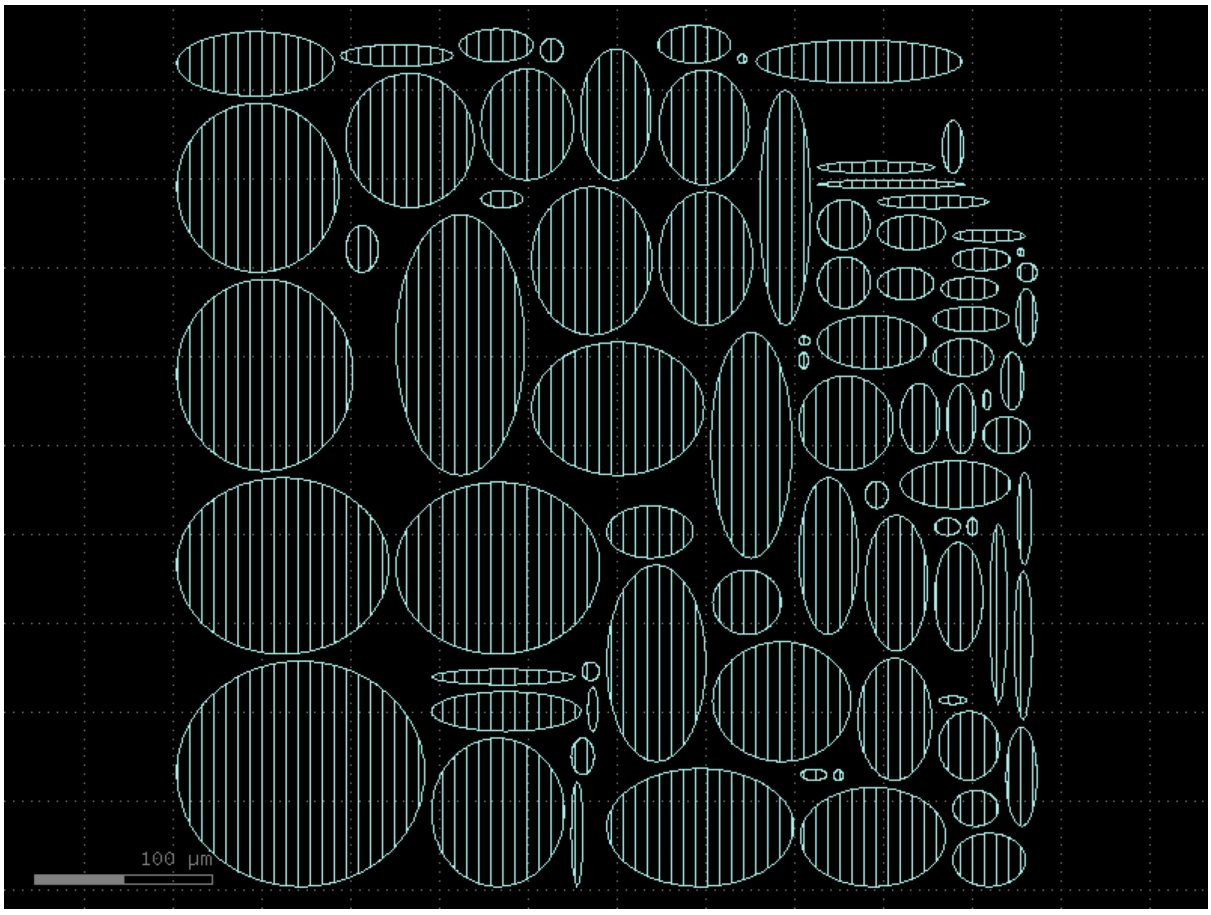
2.10 qpdk.samples.sample5.sample5_path

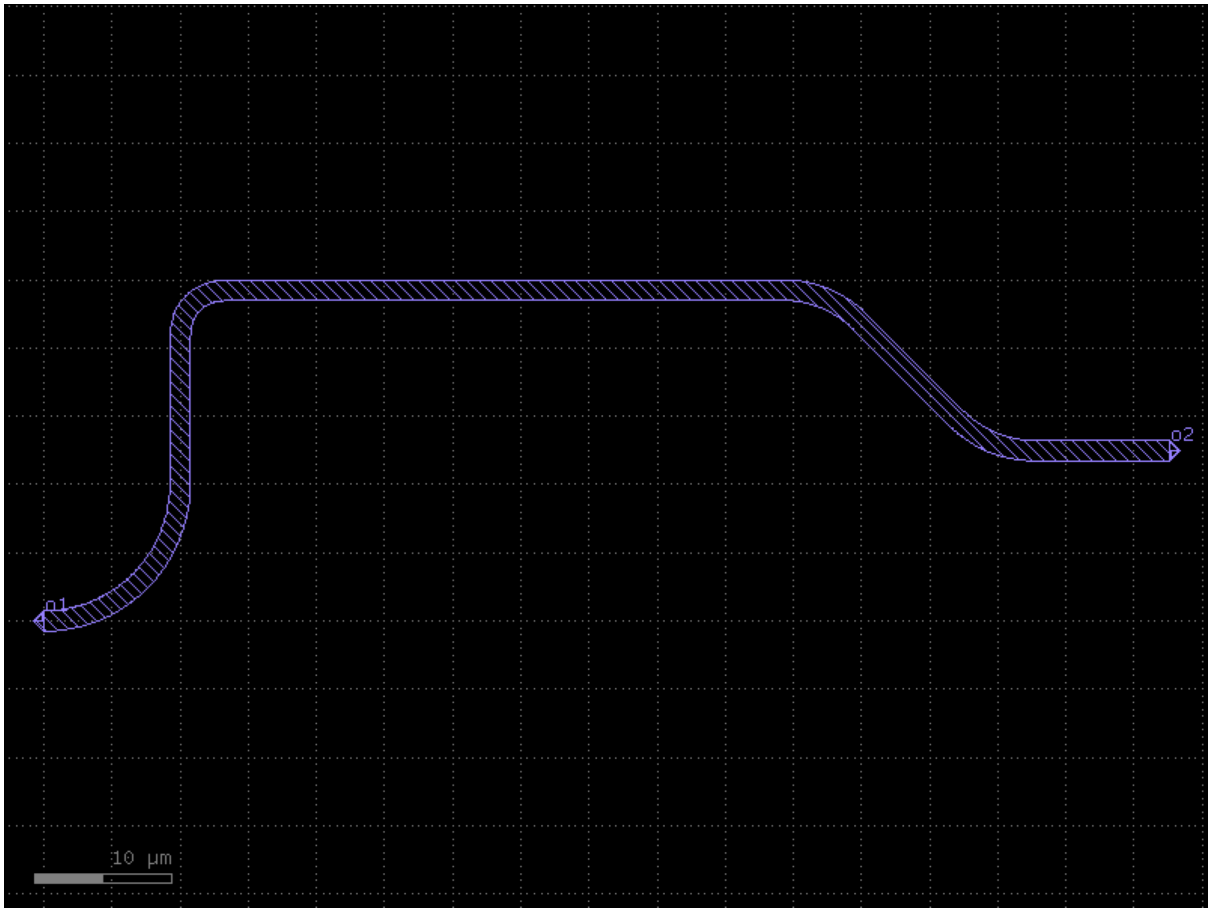
qpdk.samples.sample5.**sample5_path**()

Returns a component with a path made of different segments.

```
import qpdk.samples.sample5
from qpdk import PDK

PDK.activate()
c = qpdk.samples.sample5.sample5_path().copy()
c.draw_ports()
c.plot()
```


100 µm


o1
o2
10 µm

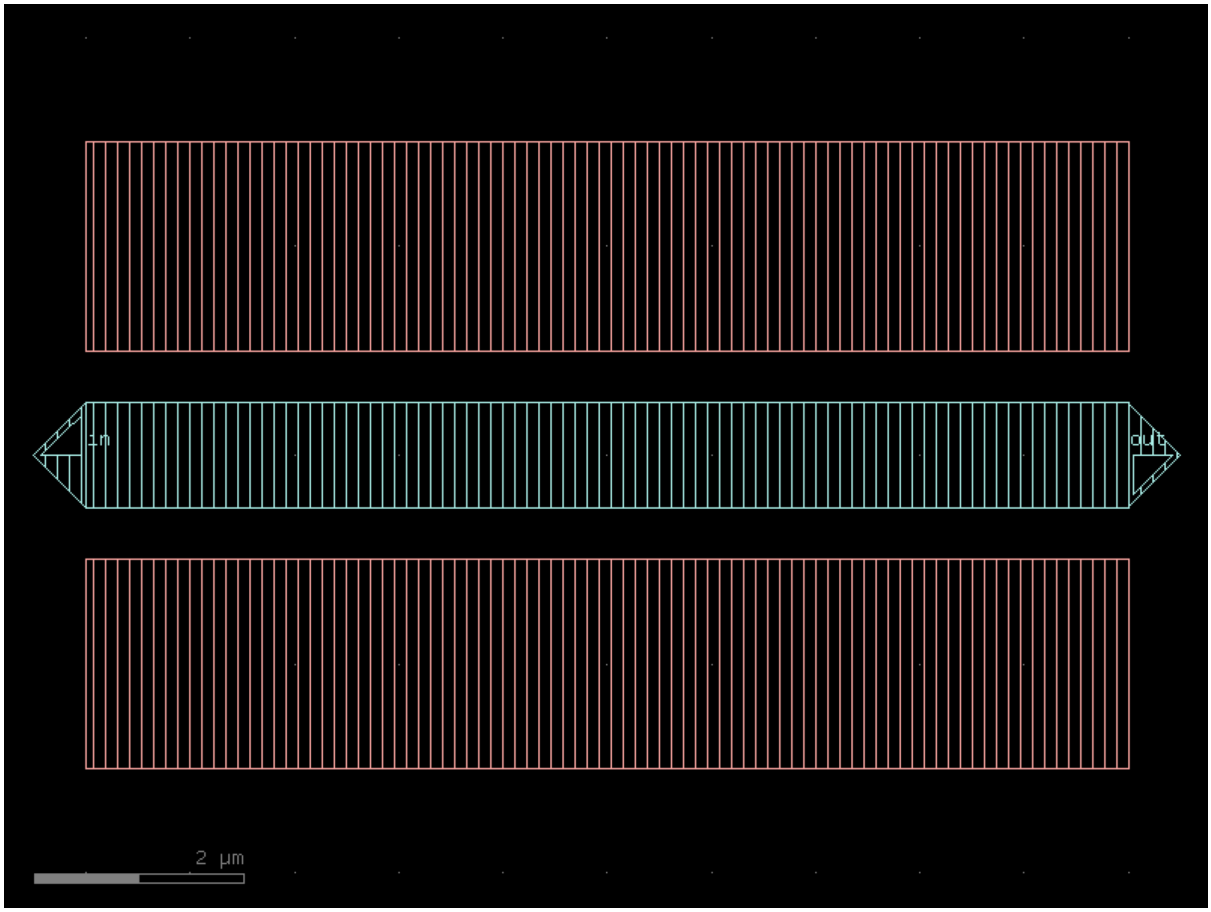
2.11 qpdk.samples.sample6.sample6_cross_section

qpdk.samples.sample6.**sample6_cross_section**()

Returns a component with a path made of different segments.

```
import qpdk.samples.sample6
from qpdk import PDK

PDK.activate()
c = qpdk.samples.sample6.sample6_cross_section().copy()
c.draw_ports()
c.plot()
```



2.12 qpdk.samples.simulate_resonator.resonator_simulation

qpdk.samples.simulate_resonator.**resonator_simulation**(*coupling_gap=12.0*)

Create a resonator simulation layout with launchers and CPW routes.

Parameters

coupling_gap (*float*)

Return type

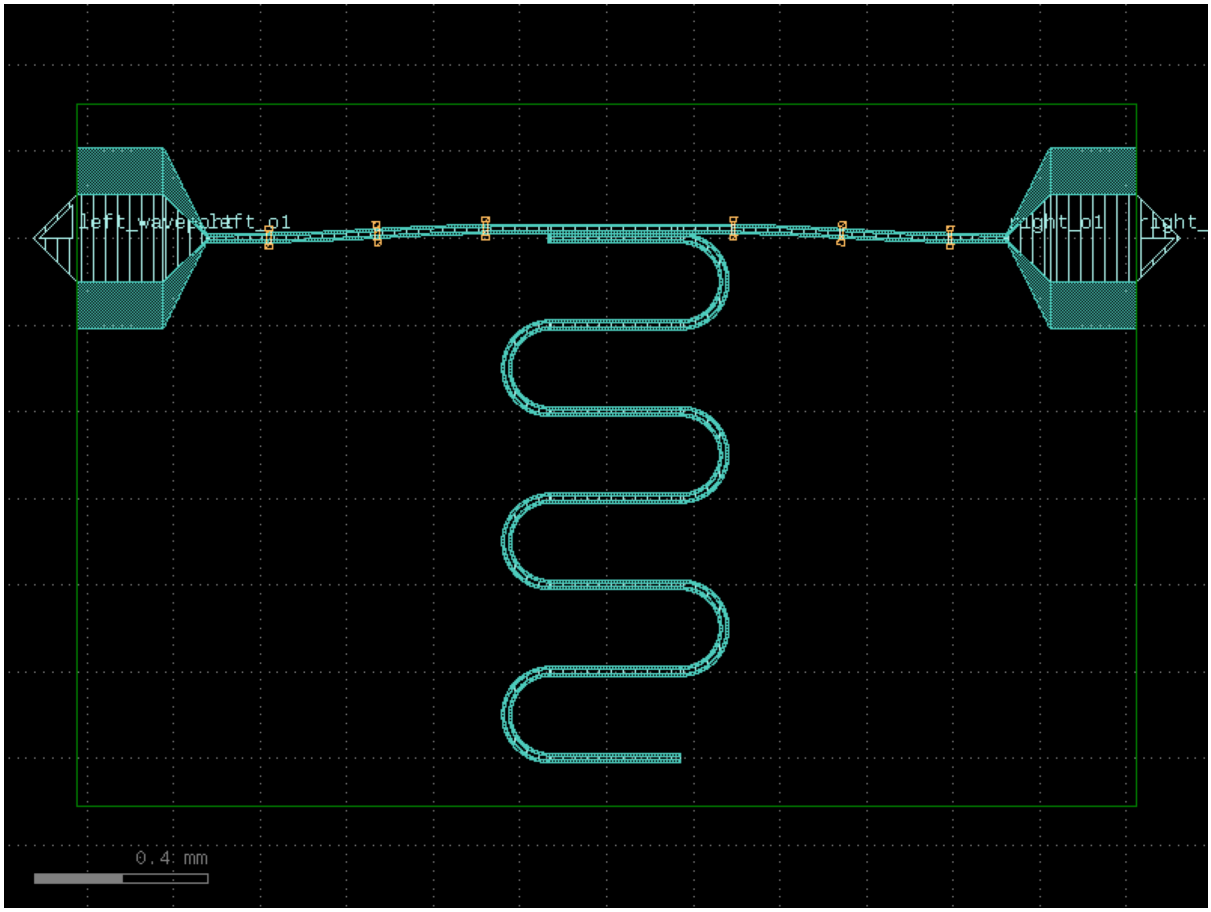
Component

```
import qpdk.samples.simulate_resonator
from qpdk import PDK
```

(continues on next page)

(continued from previous page)

```
PDK.activate()
c = qpdk.samples.simulate_resonator.resonator_simulation(coupling_gap=12).copy()
c.draw_ports()
c.plot()
```



3

References

Part III

Notebooks

4 Notebooks

These notebooks demonstrate integration to other relevant tools for design and simulations.

- Table of Contents
  - *Optuna Optimization of Interdigital Capacitor* (page 63)
  - *Resonator frequency estimation models* (page 68)
  - *Superconducting Qubit Parameter Calculations with scqubits* (page 71)

4.1 Optuna Optimization of Interdigital Capacitor

This example demonstrates using Optuna[24] to optimize an interdigital capacitor to achieve a target capacitance of 40 fF. The optimization is constrained to use exactly 5 interdigital fingers.

4.1.1 Simulation Layout Setup

Create a simulation layout with an interdigital capacitor, some extended straights and an etch for capacitive simulation.

```
@gf.cell
def capacitor_simulation(
    finger_length: float = 20.0,
    finger_gap: float = 2.0,
    thickness: float = 5.0,
) -> gf.Component:
    """Create a capacitor simulation layout with launchers and direct connections.

    Args:
        finger_length: Length of each finger in µm.
        finger_gap: Gap between adjacent fingers in µm.
        thickness: Thickness of fingers and base section in µm.

    Returns:
        Component with the simulation layout including ports.
    """
    c = gf.Component()

    # Create interdigital capacitor with fixed 5 fingers as specified
    cap_ref = c << interdigital_capacitor(
        fingers=5,  # Fixed to 5 fingers as per requirements
        finger_length=finger_length,
```

(continues on next page)

[24] https://optuna.readthedocs.io/en/stable/index.html

(continued from previous page)

```
        finger_gap=finger_gap,
        thickness=thickness,
        etch_layer="M1_ETCH",
        etch_bbox_margin=5.0,
        cross_section="cpw",
        half=False,
    )

    # Add straights for larger area
    straight_left = c << straight(length=20.0, cross_section="cpw")
    straight_right = c << straight(length=20.0, cross_section="cpw")
    straight_left.connect("o2", cap_ref.ports["o1"])
    straight_right.connect("o1", cap_ref.ports["o2"])

    # Add etched end at the straights
    etch_left = c << straight(length=6.0, cross_section="etch")
    etch_right = c << straight(length=6.0, cross_section="etch")
    etch_left.connect("o2", straight_left.ports["o1"], allow_layer_mismatch=True)
    etch_right.connect("o1", straight_right.ports["o2"], allow_layer_
↪mismatch=True)

    # Add simulation area layer around the layout
    c.kdb_cell.shapes(LAYER.SIM_AREA).insert(c.bbox().enlarged(50, 50))

    # Add ports for marking capacitor terminals
    c.add_port(name="M1_left", port=straight_left.ports["o1"])
    c.add_port(name="M1_right", port=straight_right.ports["o2"])

    return c
```

4.1.2 Optuna Objective Function

Define the objective function that Optuna will optimize. The goal is to minimize the difference between the simulated capacitance and the target of 40 fF.

```
def _objective_function(trial: optuna.trial.Trial) -> float:
    """Optuna objective function to optimize capacitor for target capacitance.

    Args:
        trial: Optuna trial object with parameter suggestions.

    Returns:
        Objective value (difference from target capacitance).
    """
    target_capacitance = 40.0  # in fF

    # Define parameter search space
    finger_length = trial.suggest_float("finger_length", 5.0, 50.0)  # µm
    finger_gap = trial.suggest_float("finger_gap", 1.0, 10.0)  # µm
    thickness = trial.suggest_float("thickness", 2.0, 10.0)  # µm

    try:
        c = capacitor_simulation(
            finger_length=finger_length,
            finger_gap=finger_gap,
            thickness=thickness,
        )
        simulated_capacitance = _run_capacitive_simulation(c)
```

(continues on next page)

(continued from previous page)

```
        # Calculate objective: minimize mean squared error from target capacitance
        objective_value = (simulated_capacitance - target_capacitance) ** 2

        # Store additional info for analysis
        trial.set_user_attr("simulated_capacitance", simulated_capacitance)
        trial.set_user_attr("target_capacitance", target_capacitance)

        return objective_value

    except Exception as e:
        print(f"Trial failed with error: {e}")
        return 1000.0  # Large penalty value
```

4.1.3 Palace imulation settings

This section shows how the Palace simulation is set up.

```
def _setup_palace_simulation() -> dict[str, Any]:
    """Setup configuration for Palace capacitive simulation.

    The mesh_parameters section is provided as keyword arguments
    to :func:`~meshwell.mesh.mesh`.

    The mesh parameters here are not optimized but serve as a reasonable
    starting point while demonstrating how te set up the mesh in different ways.

    Args:
        component: The gdsfactory component to simulate.

    Returns:
        Dictionary with simulation configuration.
    """
    simulation_folder = PATH.simulation / "capacitor_simulation"
    simulation_folder.mkdir(exist_ok=True)

    # Palace simulation configuration
    return {
        "layer_stack": PDK.layer_stack,
        "material_spec": material_properties,
        "simulation_folder": simulation_folder,
        "mesh_parameters": {
            "default_characteristic_length": 30,  # µm
            "resolution_specs": {
                "M1@M1_left": [
                    ExponentialField(
                        sizemin=0.3, lengthscale=2, growth_factor=2.0, apply_to=
↪"curves"
                    ),
                    ConstantInField(resolution=0.3, apply_to="surfaces"),
                ],
                "M1@M1_right": [
                    ExponentialField(
                        sizemin=0.3, lengthscale=2, growth_factor=2.0, apply_to=
↪"curves"
                    ),
                    ConstantInField(resolution=0.3, apply_to="surfaces"),
                ],
                "M1": [ConstantInField(resolution=8.0, apply_to="volumes")],
                "Substrate": [
```

(continues on next page)

(continued from previous page)

```
                    ConstantInField(resolution=5.0, apply_to="curves"),
                    ConstantInField(resolution=8.0, apply_to="surfaces"),
                    ConstantInField(resolution=15.0, apply_to="volumes"),
                ],
                "Vacuum": [
                    ConstantInField(resolution=5.0, apply_to="curves"),
                    ConstantInField(resolution=15.0, apply_to="surfaces"),
                    ConstantInField(resolution=25.0, apply_to="volumes"),
                ],
            },
            "verbosity": 10,
        },
    }


def _run_capacitive_simulation(component: gf.Component) -> float:
    """Run Palace capacitive simulation (requires system dependencies)."""
    from gplugins.palace import run_capacitive_simulation_palace

    config = _setup_palace_simulation()
    results = run_capacitive_simulation_palace(component, n_processes=4, **config)
    return results.capacitance_matrix[tuple(p.name for p in component.ports)] *␣
↪1e15
```

4.1.4 Main Execution and Optuna Study

Run the optimization study using Optuna to find parameters that achieve the target capacitance of 40 fF.

This section is not run in the documentation because Palace requires an installation and the optimization may take time.

```
if __name__ == "__main__":
    from qpdk import PDK

    PDK.activate()

    # First ensure a single simulation runs correctly
    c = capacitor_simulation()
    c.show()
    simulated_capacitance = _run_capacitive_simulation(c)
    print(f"Single simulation capacitance: {simulated_capacitance:.3f} fF")

    # Create an Optuna study for minimization
    study = optuna.create_study(
        direction="minimize",
        study_name="interdigital_capacitor_optimization",
    )

    print("Starting Optuna optimization for 40 fF interdigital capacitor...")
    print("Target: 40 fF capacitance with 5 fingers")
    print("Optimizing: finger_length, finger_gap, thickness")
    print()
    study.optimize(_objective_function, n_trials=5, n_jobs=1, show_progress_
↪bar=True)

    # Check if we have any successful trials
    successful_trials = [t for t in study.trials if t.value < 1000.0]

    if successful_trials:
```

(continues on next page)

(continued from previous page)

```
        print(f"Best trial: {study.best_trial.number}")
        print(f"Best objective value: {study.best_value:.3f} fF difference from␣
↪target")
        print("\nBest parameters:")
        for param, value in study.best_trial.params.items():
            print(f"  {param}: {value:.3f} µm")

        if "simulated_capacitance" in study.best_trial.user_attrs:
            print(
                f"\nSimulated capacitance: {study.best_trial.user_attrs['simulated_
↪capacitance']:.3f} fF"
            )
            print(
                f"Target capacitance: {study.best_trial.user_attrs['target_
↪capacitance']:.3f} fF"
            )

        # Create and show the optimized component
        best_params = study.best_trial.params
        optimized_component = capacitor_simulation(
            finger_length=best_params["finger_length"],
            finger_gap=best_params["finger_gap"],
            thickness=best_params["thickness"],
        )

        print("\nOptimized component created with:")
        print("  - 5 fingers (fixed)")
        print(f"  - finger_length: {best_params['finger_length']:.3f} µm")
        print(f"  - finger_gap: {best_params['finger_gap']:.3f} µm")
        print(f"  - thickness: {best_params['thickness']:.3f} µm")
    else:
        print(
            "No successful trials found. All trials resulted in component␣
↪generation errors."
        )
        print(
            "This suggests there may be issues with the component geometry or␣
↪routing."
        )
        # Still try to create a component with default parameters for debugging
        print("\nTrying to create component with default parameters for debugging..
↪.")
        try:
            test_component = capacitor_simulation()
            print("Default component created successfully")
        except Exception as e:
            print(f"Error creating default component: {e}")

    # Display the component (optional - requires display backend)
    try:
        if successful_trials:
            optimized_component.show()
        else:
            test_component.show()
    except Exception:
        print("(Component display not available in this environment)")

    # Save optimization history for analysis
    if not PATH.simulation.exists():
        PATH.simulation.mkdir()
    results_file = PATH.simulation / "capacitor_optimization_results.csv"
```

(continues on next page)

(continued from previous page)

```
    study.trials_dataframe().to_csv(results_file, index=False)
    print(f"\nOptimization results saved to: {results_file}")
```

4.2 Resonator frequency estimation models

This example demonstrates estimating resonance frequencies of superconducting microwave resonators using scikit-rf and Jax.

4.2.1 Probelines weakly coupled to $\lambda/4$ resonator

Creates a probelines weakly coupled to a quarter-wave resonator. The resonance frequency is first estimated using the `resonator_frequency` function and then compared to the frequency in the coupled case.

```
if __name__ == "__main__":
    import matplotlib.pyplot as plt

    cpw = cpw_media_skrf(width=10, gap=6)(
        frequency=skrf.Frequency(2, 9, 101, unit="GHz")
    )
    print(f"{cpw=!r}")
    print(f"{cpw.z0.mean().real=!r}")  # Characteristic impedance

    res_freq = resonator_frequency(length=4000, media=cpw, is_quarter_wave=True)
    print("Resonance frequency (quarter-wave):", res_freq / 1e9, "GHz")

    circuit, info = sax.circuit(
        netlist={
            "instances": {
                "R1": "quarter_wave_resonator",
            },
            "connections": {},
            "ports": {
                "in": "R1,o1",
                "out": "R1,o2",
            },
        },
        models={
            "quarter_wave_resonator": partial(
                quarter_wave_resonator_coupled_to_probeline,
                media=cpw_media_skrf(width=10, gap=6),
                length=4000,
                coupling_capacitance=15e-15,
            )
        },
    )

    frequencies = jnp.linspace(1e9, 10e9, 5001)
    S = circuit(f=frequencies)
    print(info)
    plt.plot(frequencies / 1e9, abs(S["in", "out"]) ** 2)
    plt.xlabel("f [GHz]")
    plt.ylabel("$S_{21}$")

    def _mark_resonance_frequency(x_value: float, color: str, label: str):
        """Draws a vertical dashed line on the current matplotlib plot to mark a␣
↪resonance frequency."""
```

(continues on next page)

(continued from previous page)

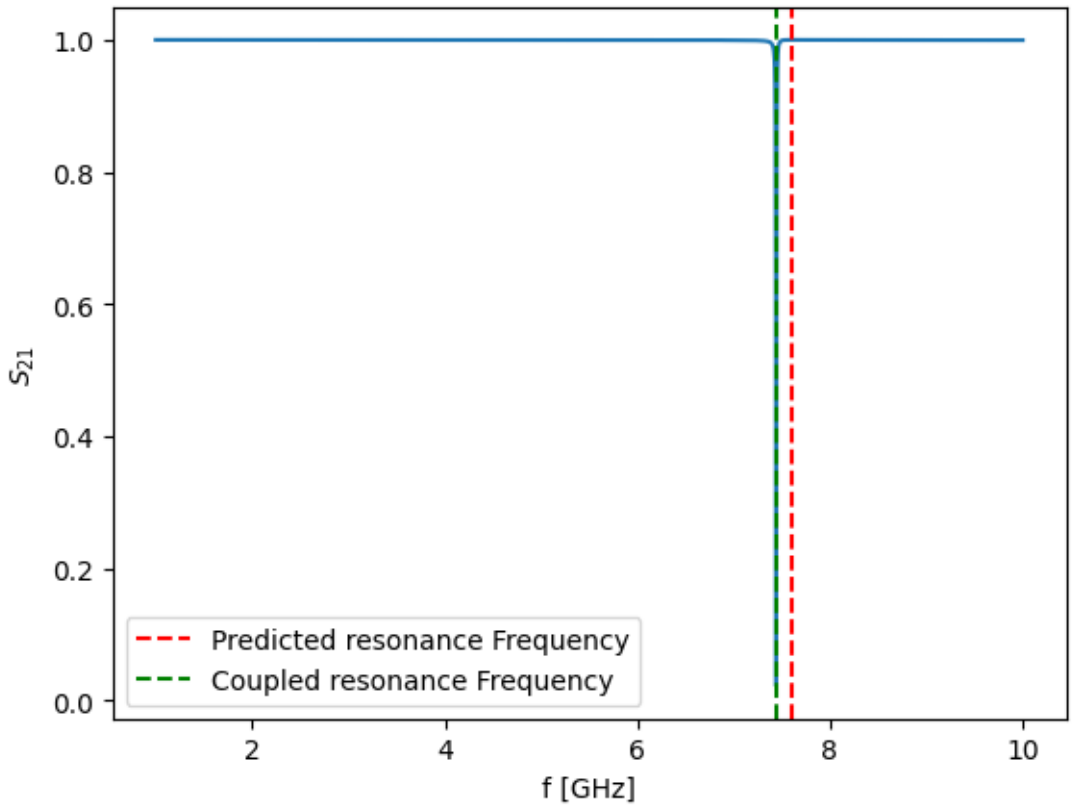
```
        plt.axvline(
            x_value / 1e9,  # Convert frequency from Hz to GHz for plotting
            color=color,
            linestyle="--",
            label=label,
        )

    _mark_resonance_frequency(res_freq, "red", "Predicted resonance Frequency")
    actual_freq = frequencies[jnp.argmin(abs(S["in", "out"]))]
    print("Coupled resonance frequency:", actual_freq / 1e9, "GHz")
    _mark_resonance_frequency(actual_freq, "green", "Coupled resonance Frequency")

    plt.legend()
    # plt.show()
```

```
cpw=Coplanar Waveguide Media.  2.0-9.0 GHz. 101 points
 W= 1.00e-05m, S= 6.00e-06m
cpw.z0.mean().real=np.float64(49.27868570939533)
Resonance frequency (quarter-wave): 7.6081394901138095 GHz
```

```
CircuitInfo(dag=<networkx.classes.digraph.DiGraph object at 0x7f7d12c4cd40>,␣
 ↪models={'quarter_wave_resonator': functools.partial(<function quarter_wave_
 ↪resonator_coupled_to_probeline at 0x7f7d12cdcb80>, media=functools.partial(
 ↪<class 'skrf.media.cpw.CPW'>, w=9.999999999999999e-06, s=6e-06, h=0.0005, t=1.
 ↪9999999999999996e-07, ep_r=11.45, rho=1e-100, tand=0), length=4000, coupling_
 ↪capacitance=1.5e-14), 'top_level': <function _flat_circuit.<locals>._circuit␣
 ↪at 0x7f7d111449a0>}, backend='klu')
Coupled resonance frequency: 7.4404 GHz
```



4.2.2 Optimizer for given resonance frequency

Find the resonator length that gives a desired resonance frequency using an optimizer.

```
if __name__ == "__main__":
    import ray
    import ray.tune
    import ray.tune.search.optuna

    frequencies = jnp.linspace(0.5e9, 10e9, 1001)
    TARGET_FREQUENCY = 6e9  # Target resonance frequency in Hz

    def loss_fn(config: dict[str, float]) -> float:
        """Loss function to minimize the difference between the actual and target resonance frequencies.

        Args:
            config: Dictionary containing the resonator length in micrometers.
        """
        length = config["length"]
        # Setup model
        S = circuit(f=frequencies, length=length)
        # Get frequency at minimum S21
        coupled_freq = frequencies[jnp.argmin(abs(S["in", "out"]))]
        return {
            "l1_loss_ghz": abs(float(coupled_freq) - TARGET_FREQUENCY) / 1e9,
            "mse": (float(coupled_freq) - TARGET_FREQUENCY) ** 2,
        }

    # Test loss function
    print(f"{loss_fn(dict(length=4000.0))=}")
    print(f"{loss_fn(dict(length=5900.0))=}")

    # Optimize length using Ray Tune
    tuner = ray.tune.Tuner(
        loss_fn,
        param_space={
            "length": ray.tune.uniform(1000.0, 9000.0),
        },
        tune_config=ray.tune.TuneConfig(
            metric="mse",
            mode="min",
            num_samples=10,
            max_concurrent_trials=math.ceil(os.cpu_count() / 4),
            reuse_actors=True,
            search_alg=ray.tune.search.optuna.OptunaSearch(),
        ),
    )
    results = tuner.fit()
    best_trial = results.get_best_result()
    length = best_trial.config["length"]
    print(f"Best trial config: {best_trial.config}")

    # Initialize optimizer
    print(f"Optimized Length: {length:.2f} µm")
    optimal_S = circuit(f=frequencies, length=length)
    optimal_freq = frequencies[jnp.argmin(abs(optimal_S["in", "out"]))]
    print(f"Achieved Resonance Frequency: {optimal_freq / 1e9:.2f} GHz")

    # Plot
    plt.plot(frequencies / 1e9, abs(optimal_S["in", "out"]) ** 2)
```

(continues on next page)

(continued from previous page)

```
    plt.xlabel("f [GHz]")
    plt.ylabel("$S_{21}$")
    _mark_resonance_frequency(optimal_freq, "blue", "Optimized resonance Frequency
↪")
    _mark_resonance_frequency(TARGET_FREQUENCY, "orange", "Target resonance␣
↪Frequency")
    plt.legend()
    plt.show()
```

4.3 Superconducting Qubit Parameter Calculations with scqubits

This example demonstrates how to use scqubits[25] [GK21] to calculate parameters for superconducting qubits and coupled resonators. The calculations help determine physical parameters like capacitances and coupling strengths needed for component design.

4.3.1 TunableTransmon Qubit Parameters

A tunable transmon qubit [KYG+07] has a SQUID junction that allows tuning the effective Josephson energy through an external flux. Key parameters include:

- $E_{\mathrm{J_{max}}}$: Maximum Josephson energy when flux = 0
- E_{C}: Charging energy, related to total capacitance as $E_{\mathrm{C}} = \dfrac{e^2}{2C_{\Sigma}}$
- d: Asymmetry parameter between the two junctions in the SQUID
- ϕ: External flux in units of flux quantum ϕ_0
- α: Anharmonicity, the difference between the $E_{1\to2}$ and $E_{0\to1}$ transition frequencies

See the scqubits documentation[26] for more details.

```
# Create a tunable transmon with realistic parameters
transmon = scq.TunableTransmon(
    EJmax=40.0,  # Maximum Josephson energy in GHz (typical range: 20-50 GHz)
    EC=0.2,  # Charging energy in GHz (typical range: 0.1-0.5 GHz)
    d=0.1,  # Junction asymmetry (typical range: 0.05-0.2)
    flux=0.0,  # Start at flux = 0 (maximum EJ)
    ng=0.0,  # Offset charge (usually 0 or 0.5)
    ncut=30,  # Charge basis cutoff
)

# Tunable Transmon Parameters
display(
    Math(rf"""
\textbf{{Tunable Transmon Parameters:}} \\
E_{{J,\text{{max}}}} = {transmon.EJmax:.1f}\ \text{{GHz}} \\
E_C = {transmon.EC:.1f}\ \text{{GHz}} \\
d = {transmon.d:.2f} \\
\text{{Flux}} = {transmon.flux:.2f}\ \Phi_0
""")
)

# Calculate energy levels
eigenvals = transmon.eigenvals(evals_count=5)
```

(continues on next page)

[25] https://scqubits.readthedocs.io/en/latest/
[26] https://scqubits.readthedocs.io/en/latest/guide/qubits/tunable_transmon.html

(continued from previous page)

```
f01 = eigenvals[1] - eigenvals[0]
f12 = eigenvals[2] - eigenvals[1]
anharmonicity = f12 - f01

display(
    Math(rf"""
\textbf{{Transmon Spectrum:}} \\
0\rightarrow 1\ \text{{frequency}}:\ \ {f01:.3f}\ \text{{GHz}} \\
1\rightarrow 2\ \text{{frequency}}:\ \ {f12:.3f}\ \text{{GHz}} \\
\text{{Anharmonicity, }} \alpha:\ \ {anharmonicity:.3f}\ \text{{GHz}}
""")
)
```

Tunable Transmon Parameters:

$$E_{J,\max} = 40.0 \text{ GHz}$$
$$E_C = 0.2 \text{ GHz}$$
$$d = 0.10$$
$$\text{Flux} = 0.00\ \Phi_0$$

Transmon Spectrum:

$$0 \rightarrow 1 \text{ frequency}: \quad 7.795 \text{ GHz}$$
$$1 \rightarrow 2 \text{ frequency}: \quad 7.582 \text{ GHz}$$
$$\text{Anharmonicity, } \alpha: \quad -0.213 \text{ GHz}$$

4.3.2 Parameter Sweep: Flux Dependence

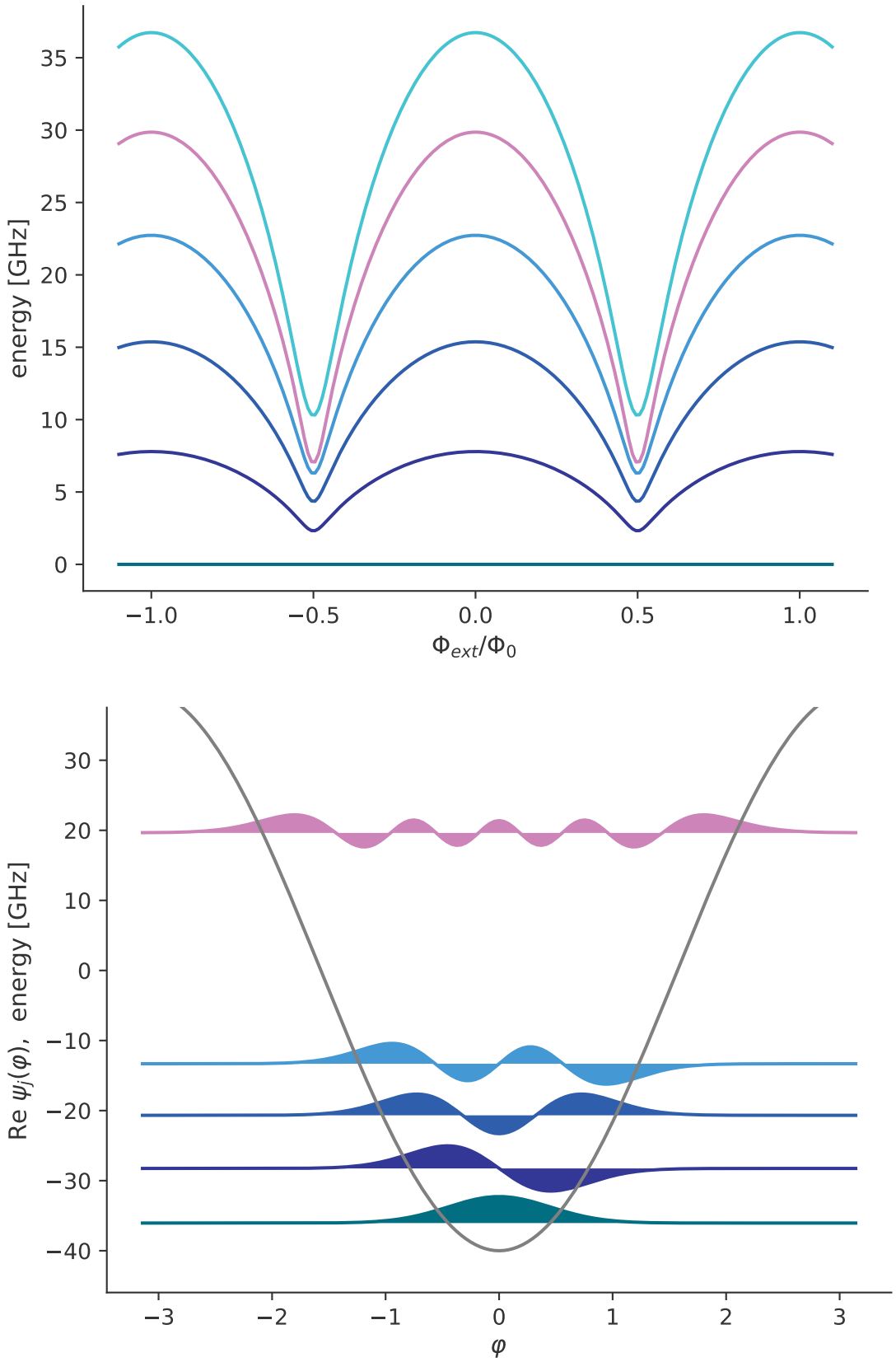
Demonstrate how the qubit frequency changes with external flux.

For more details, see the scqubits-examples repository[27]

```
# Sweep flux and calculate qubit frequency
transmon.plot_evals_vs_paramvals(
    "flux", np.linspace(-1.1, 1.1, 201), subtract_ground=True
)
plt.show()
transmon.plot_wavefunction(esys=None, which=(0, 1, 2, 3, 8), mode="real")
plt.show()
```

```
Spectral data:   0%|          | 0/201 [00:00<?, ?it/s]
```

[27] https://github.com/scqubits/scqubits-examples


energy [GHz]
35
30
25
20
15
10
5
0
−1.0
−0.5
0.0
0.5
1.0
Φ_{ext}/Φ_0
Re $\psi_j(\varphi)$, energy [GHz]
30
20
10
0
−10
−20
−30
−40
−3
−2
−1
0
1
2
3
φ

4.3.3 Coupled Qubit-Resonator System

When a transmon is coupled to a resonator, the interaction can be described by the Jaynes-Cummings model [SK93] :

$$\mathcal{H} = \omega_r a^\dagger a + \frac{\omega_q}{2}\sigma_z + g(a^\dagger \sigma^- + a\sigma^+), \tag{4.1}$$

where ω_r is the resonator frequency, ω_q is the qubit frequency, and g is the coupling strength. The operators $a^\dagger$ and a are the creation and annihilation operators for the resonator, while σ_z, σ^-, and σ^+ are the Pauli and ladder operators for the qubit.

The coupling strength g depends on the overlap between the qubit and resonator modes and affects the system dynamics.

```
# Resonator (Oscillator) Parameters
resonator = scq.Oscillator(
    E_osc=6.5,  # Resonator frequency in GHz (typical range: 4-8 GHz)
)

display(
    Math(rf"""
\textbf{{Resonator Parameters:}} \\
\text{{Frequency}}:\ \ {resonator.E_osc:.1f}\ \text{{GHz}} \\
""")
)

# Create a coupled system using HilbertSpace
hilbert_space = scq.HilbertSpace([transmon, resonator])

# Add interaction between qubit and resonator
# The interaction is typically g(n_qubit * (a + a†)) where n is the charge␣
 ↪operator and a is the resonator annihilation operator
g_coupling = 0.15  # Coupling strength in GHz (typical range: 0.05-0.3 GHz)
interaction = scq.InteractionTerm(
    g_strength=g_coupling,
    operator_list=[
        (0, transmon.n_operator),  # (subsystem_index, operator)
        (1, resonator.annihilation_operator() + resonator.creation_operator()),
    ],
)
hilbert_space.interaction_list = [interaction]


display(Math(f"g_{{\\text{{Qubit-Resonator}}}} = {g_coupling:.3f}\\,\\text{{GHz}}
 ↪"))

# Calculate the coupled system eigenvalues
coupled_eigenvals = hilbert_space.eigenvals(evals_count=10)
print(f"  Coupled system ground state: {coupled_eigenvals[0]:.3f} GHz")
```

Resonator Parameters:

Frequency : 6.5 GHz

$$g_{\text{Qubit–Resonator}} = 0.150\,\text{GHz}$$

```
  Coupled system ground state: -36.054 GHz
```

4.3.4 Physical (Lumped-element) Parameter Extraction

From the quantum model, we can extract physical parameters relevant for PDK design.

The coupling strength g (in the dispersive limit $g \ll \omega_q, \omega_r$) can be related to a coupling capacitance C_c via [Sav23]:

where C_Σ is the total qubit capacitance, C_q is the capacitance between the qubit pads, and C_r is the total capacitance of the resonator.

```
# Calculate physical capacitance
# The charging energy EC = e²/(2C_total), so C_Σ = e²/(2*EC)
EC_joules = transmon.EC * 1e9 * scipy.constants.h  # Convert GHz to Joules

# Total capacitance of the transmon
C_Σ = scipy.constants.e**2 / (2 * EC_joules)

# Josephson inductance - use typical realistic value
# For EJ ~ 40 GHz, typical LJ ~ 0.8-1.0 nH
# LJ scales inversely with EJ: LJ ~ 1.0 nH * (25 GHz / EJ)
LJ_typical = 1.0e-9 * (25.0 / transmon.EJmax)  # Typical scaling

# Compute coupling capacitance from coupling strength
C_c_sym, C_Σ_sym, C_r_sym, omega_q_sym, omega_r_sym, g_sym = sp.symbols(
    "C_c C_Σ C_r omega_q omega_r g", real=True, positive=True
)
equation = sp.Eq(
    g_sym,
    0.5 * (C_c_sym / sp.sqrt(C_Σ_sym * C_r_sym)) * sp.sqrt(omega_q_sym * omega_r_
 ↪sym),
)
solution = sp.solve(equation, C_c_sym)
C_c_sol = next(sol for sol in solution if sol.is_real and sol > 0)
display(Math(f"C_\\text{{c}} = {sp.latex(C_c_sol)}"))


# Use a typical value for resonator capacitance to ground
resonator_media = cpw_media_skrf(width=10, gap=6)(
    frequency=skrf.Frequency.from_f([5], unit="GHz")
)


def _objective(length: float) -> float:
    """Find resonator length for target frequency using SciPy."""
    freq = resonator_frequency(
        length=length, media=resonator_media, is_quarter_wave=True
    )
    return (freq - resonator.E_osc * 1e9) ** 2  # MSE


length_initial = 4000.0  # Initial guess in µm
result = scipy.optimize.minimize(_objective, length_initial, bounds=[(1000,␣
 ↪20000)])
length = result.x[0]
print(
    f"Optimization success: {result.success}, message: {result.message}, nfev:
 ↪{result.nfev}"
)
display(
    Math(
        f"\\textrm{{Resonator length at width 10 µm and gap 6 µm}}\\,{resonator.E_
 ↪osc:.1f}\\,\\mathrm{{GHz}}: {length:.1f}\\,\\mathrm{{µm}}"
    )
)
```

$$C_c = \frac{2.0\sqrt{C_r}\sqrt{C_\Sigma}g}{\sqrt{\omega_q}\sqrt{\omega_r}}$$

```
Optimization success: True, message: CONVERGENCE: RELATIVE REDUCTION OF F <=␣
↪FACTR*EPSMCH, nfev: 24
```

Resonator length at width 10 µm and gap 6 µm 6.5 GHz : 4681.9 µm

The total capacitance of the resonator can be estimated from its characteristic impedance Z_0 and phase velocity v_p as [GopplFB+08]:

$$C_r = \frac{l}{\mathrm{Re}(Z_0 v_p)} \tag{4.2}$$

where l is the resonator length.

```
# Get total capacitance to ground of the resonator (in isolation, we disregard␣
↪effect of coupling to qubits)
C_r = 1 / np.real(resonator_media.z0 * resonator_media.v_p).mean() * length * 1e-
↪6  # F

# Substitute and evaluate numerically
C_c_num = C_c_sol.subs(
    {
        g_sym: g_coupling,
        C_Σ_sym: C_Σ,
        C_r_sym: C_r,
        omega_q_sym: f01,
        omega_r_sym: resonator.E_osc,
    }
).evalf()

display(
    Math(rf"""
\textbf{{Physical Parameters for PDK Design:}}\\
\text{{Total qubit capacitance:}}~{C_Σ * 1e15:.1f}~\mathrm{{fF}}\\
\text{{Josephson inductance:}}~{LJ_typical * 1e9:.2f}~\mathrm{{nH}}\\
\text{{Estimated target qubit-resonator coupling capacitance:}}~{C_c_num * 1e15:.
↪1f}~\mathrm{{fF}}
""")
)
```

Physical Parameters for PDK Design:

Total qubit capacitance: 96.9 fF

Josephson inductance: 0.62 nH

Estimated target qubit–resonator coupling capacitance: 11.6 fF

4.3.5 References

Bibliography

[BKM+13] R. Barends, J. Kelly, A. Megrant, D. Sank, E. Jeffrey, Y. Chen, Y. Yin, B. Chiaro, J. Mutus, C. Neill, P. O'Malley, P. Roushan, J. Wenner, T. C. White, A. N. Cleland, and John M. Martinis. Coherent Josephson Qubit Suitable for Scalable Quantum Integrated Circuits. *Physical Review Letters*, 111(8):080502, August 2013. URL: https://link.aps.org/doi/10.1103/PhysRevLett.111.080502 (visited on 2025-09-06), doi:10.1103/PhysRevLett.111.080502[13].

[BM18] Ilya Besedin and Alexey P Menushenkov. Quality factor of a transmission line coupled coplanar waveguide resonator. *EPJ Quantum Technology*, 5(1):2, December 2018. URL: https://epjquantumtechnology.springeropen.com/articles/10.1140/epjqt/s40507-018-0066-3 (visited on 2025-09-10), doi:10.1140/epjqt/s40507-018-0066-3[14].

[CB04] John Clarke and Alex I. Braginski. *The SQUID Handbook*. Wiley-VCH, Weinheim, 2004. ISBN 978-3-527-40229-8.

[KYG+07] Jens Koch, Terri M. Yu, Jay Gambetta, A. A. Houck, D. I. Schuster, J. Majer, Alexandre Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Charge-insensitive qubit design derived from the Cooper pair box. *Physical Review A*, 76(4):042319, October 2007. URL: https://link.aps.org/doi/10.1103/PhysRevA.76.042319 (visited on 2025-09-04), doi:10.1103/PhysRevA.76.042319[15].

[LWJ+25] Xuegang Li, Junhua Wang, Yao-Yao Jiang, Guang-Ming Xue, Xiaoxia Cai, Jun Zhou, Ming Gong, Zhao-Feng Liu, Shuang-Yu Zheng, Deng-Ke Ma, Mo Chen, Wei-Jie Sun, Shuang Yang, Fei Yan, Yi-Rong Jin, S. P. Zhao, Xue-Feng Ding, and Hai-Feng Yu. Cosmic-ray-induced correlated errors in superconducting qubit array. *Nature Communications*, 16(1):4677, May 2025. URL: https://www.nature.com/articles/s41467-025-59778-z (visited on 2025-09-04), doi:10.1038/s41467-025-59778-z[16].

[LZY+21] Xuegang Li, Yingshan Zhang, Chuhong Yang, Zhiyuan Li, Junhua Wang, Tang Su, Mo Chen, Yongchao Li, Chengyao Li, Zhenyu Mi, Xuehui Liang, Chenlu Wang, Zhen Yang, Yulong Feng, Kehuan Linghu, Huikai Xu, Jiaxiu Han, Weiyang Liu, Peng Zhao, Teng Ma, Ruixia Wang, Jingning Zhang, Yu Song, Pei Liu, Ziting Wang, Zhaohua Yang, Guangming Xue, Yirong Jin, and Haifeng Yu. Vacuum-gap transmon qubits realized using flip-chip technology. *Applied Physics Letters*, 119(18):184003, November 2021. URL: https://pubs.aip.org/apl/article/119/18/184003/40593/Vacuum-gap-transmon-qubits-realized-using-flip (visited on 2025-09-04), doi:10.1063/5.0068255[17].

[MP12] David M. Pozar. *Microwave Engineering*. John Wiley & Sons, Inc., 4 edition, 2012. ISBN 978-0-470-63155-3.

[RKD+17] D. Rosenberg, D. Kim, R. Das, D. Yost, S. Gustavsson, D. Hover, P. Krantz, A. Melville, L. Racz, G. O. Samach, S. J. Weber, F. Yan, J. L. Yoder, A. J. Kerman, and W. D. Oliver. 3D integrated su-

[13] https://doi.org/10.1103/PhysRevLett.111.080502
[14] https://doi.org/10.1140/epjqt/s40507-018-0066-3
[15] https://doi.org/10.1103/PhysRevA.76.042319
[16] https://doi.org/10.1038/s41467-025-59778-z
[17] https://doi.org/10.1063/5.0068255

perconducting qubits. *npj Quantum Information*, 3(1):42, October 2017. URL: https://www.nature.com/articles/s41534-017-0044-0 (visited on 2025-09-05), doi:10.1038/s41534-017-0044-0[18].

[TSKivijarvi+25] Mikko Tuokkola, Yoshiki Sunada, Heidi Kivijärvi, Jonatan Albanese, Leif Grönberg, Jukka-Pekka Kaikkonen, Visa Vesterinen, Joonas Govenius, and Mikko Möttönen. Methods to achieve near-millisecond energy relaxation and dephasing times for a superconducting transmon qubit. *Nature Communications*, 16(1):5421, July 2025. URL: https://www.nature.com/articles/s41467-025-61126-0 (visited on 2025-08-29), doi:10.1038/s41467-025-61126-0[19].

[YSM+20] D. R. W. Yost, M. E. Schwartz, J. Mallek, D. Rosenberg, C. Stull, J. L. Yoder, G. Calusine, M. Cook, R. Das, A. L. Day, E. B. Golden, D. K. Kim, A. Melville, B. M. Niedzielski, W. Woods, A. J. Kerman, and W. D. Oliver. Solid-state qubits integrated with superconducting through-silicon vias. *npj Quantum Information*, 6(1):59, July 2020. URL: https://www.nature.com/articles/s41534-020-00289-8 (visited on 2025-09-05), doi:10.1038/s41534-020-00289-8[20].

[LeiZhuW00] Lei Zhu and Ke Wu. Accurate circuit model of interdigital capacitor and its application to design of new quasi-lumped miniaturized filters with suppression of harmonic resonance. *IEEE Transactions on Microwave Theory and Techniques*, 48(3):347–356, March 2000. URL: https://ieeexplore.ieee.org/document/826833/ (visited on 2025-09-01), doi:10.1109/22.826833[21].

[MP12] David M. Pozar. *Microwave Engineering*. John Wiley & Sons, Inc., 4 edition, 2012. ISBN 978-0-470-63155-3.

[Sim01] Rainee Simons. *Coplanar Waveguide Circuits, Components, and Systems*. Number v. 165 in Wiley Series in Microwave and Optical Engineering. Wiley Interscience, New York, 2001. ISBN 978-0-471-22475-4 978-0-471-46393-1. doi:10.1002/0471224758[22].

[TSKivijarvi+25] Mikko Tuokkola, Yoshiki Sunada, Heidi Kivijärvi, Jonatan Albanese, Leif Grönberg, Jukka-Pekka Kaikkonen, Visa Vesterinen, Joonas Govenius, and Mikko Möttönen. Methods to achieve near-millisecond energy relaxation and dephasing times for a superconducting transmon qubit. *Nature Communications*, 16(1):5421, July 2025. URL: https://www.nature.com/articles/s41467-025-61126-0 (visited on 2025-08-29), doi:10.1038/s41467-025-61126-0[23].

[GK21] Peter Groszkowski and Jens Koch. Scqubits: a Python package for superconducting qubits. *Quantum*, 5:583, November 2021. URL: https://quantum-journal.org/papers/q-2021-11-17-583/ (visited on 2025-09-18), doi:10.22331/q-2021-11-17-583[28].

[GopplFB+08] M. Göppl, A. Fragner, M. Baur, R. Bianchetti, S. Filipp, J. M. Fink, P. J. Leek, G. Puebla, L. Steffen, and A. Wallraff. Coplanar waveguide resonators for circuit quantum electrodynamics. *Journal of Applied Physics*, 104(11):113904, December 2008. URL: https://pubs.aip.org/jap/article/104/11/113904/145728/Coplanar-waveguide-resonators-for-circuit-quantum (visited on 2025-09-18), doi:10.1063/1.3010859[29].

[KYG+07] Jens Koch, Terri M. Yu, Jay Gambetta, A. A. Houck, D. I. Schuster, J. Majer, Alexandre Blais, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Charge-insensitive qubit design derived from the Cooper pair box. *Physical Review A*, 76(4):042319, October 2007. URL: https://link.aps.org/doi/10.1103/PhysRevA.76.042319 (visited on 2025-09-04), doi:10.1103/PhysRevA.76.042319[30].

[Sav23] Niko Savola. Design and modelling of long-coherence qubits using energy participation ratios. Master's thesis, Aalto University, February 2023. URL: http://urn.fi/URN:NBN:fi:aalto-202305213270.

[SK93] Bruce W. Shore and Peter L. Knight. The Jaynes–Cummings Model. *Journal of Modern Optics*, 40(7):1195–1238, July 1993. URL: http://www.tandfonline.com/doi/abs/10.1080/09500349314551321 (visited on 2023-02-01), doi:10.1080/09500349314551321[31].

[18] https://doi.org/10.1038/s41534-017-0044-0
[19] https://doi.org/10.1038/s41467-025-61126-0
[20] https://doi.org/10.1038/s41534-020-00289-8
[21] https://doi.org/10.1109/22.826833
[22] https://doi.org/10.1002/0471224758
[23] https://doi.org/10.1038/s41467-025-61126-0
[28] https://doi.org/10.22331/q-2021-11-17-583
[29] https://doi.org/10.1063/1.3010859
[30] https://doi.org/10.1103/PhysRevA.76.042319
[31] https://doi.org/10.1080/09500349314551321

Python Module Index

q