SiPANN

Release 1.4.0

Jun 13, 2022

Getting Started

Silicon **P**hotonics with **A**rtificial **N**eural **N**etworks. SiPANN aims to implement various silicon photonics simulators based on machine learning techniques found in literature. The majority of these techniques are linear regression or neural networks. As a results SiPANN can return scattering parameters of (but not limited to)

- Half Rings
- Arbitrarily shaped directional couplers
- Racetrack Resonators
- Waveguides

And with the help of simphony and SiPANN's accompanying simphony wrapper

- Ring Resonators
- Doubly Coupled Rings
- Hybrid Devices (ie Green Machine)

CHAPTER 1

Installation

SiPANN is distributed on PyPI and can be installed with `pip`:

```
pip install SiPANN
```

1.1 Developmental Build

If you want a developmental build, it can be had by executing

```
git clone https://github.com/contagon/SiPANN.git
pip install -e SiPANN/
```

This development version allows you to make changes to this code directly (or pull changes from GitHub) without having to reinstall SiPANN each time.

You should then be able to run the examples and tutorials in the examples folder, and call SiPANN from any other python file.

Note: If installing on Windows, one of SiPANN's dependencies, `gdspy`, requires a C compiler for installation. This can be bypassed by first installing the `gdspy` wheel. This is done by downloading the wheel from gdspy, navigating to the location of the wheel, and executing

```
pip install gds*.whl
```

After this simply install SiPANN using your desired method.

CHAPTER 2

References

SiPANN is based on a variety of methods found in various papers, including:

[1] A. Hammond, E. Potokar, and R. Camacho, “Accelerating silicon photonic parameter extraction using artificial neural networks,” OSA Continuum 2, 1964-1973 (2019).

CHAPTER 3

Bibtex citation

```
@misc{SiP-ANN_2019,
        title={SiP-ANN},
        author={Easton Potokar, Alec M. Hammond, Ryan M. Camacho},
        year={2019},
        publisher={GitHub},
        howpublished={{https://github.com/contagon/SiP-ANN}}
}
```

CHAPTER 4

Releasing

Make sure you have committed a changelog file titled "[major].[minor].[patch]-changelog.md" before bumping version.

To bump version prior to a release, run one of the following commands:

```
bumpversion major
bumpversion minor
bumpversion patch
```

This will automatically create a git tag in the repository with the corrresponding version number and commit the modified files (where version numbers were updated). Pushing the tags (a manual process) to the remote will automatically create a new release. Releases are automatically published to PyPI and GitHub when git tags matching the "v*" pattern are created (e.g. "v0.2.1"), as bumpversion does.

To view the tags on the local machine, run `git tag`. To push the tags to the remote server, you can run `git push origin <tagname>`.

For code quality, please run isort and black before committing (note that the latest release of isort may not work through VSCode's integrated terminal, and it's safest to run it separately through another terminal).

4.1 Installation

SiPANN is distributed on PyPI and can be installed with `pip`:

```
pip install SiPANN
```

4.1.1 Developmental Build

If you want a developmental build, it can be had by executing

```
git clone https://github.com/contagon/SiPANN.git
pip install -e SiPANN/
```

This development version allows you to make changes to this code directly (or pull changes from GitHub) without having to reinstall SiPANN each time.

You should then be able to run the examples and tutorials in the examples folder, and call SiPANN from any other python file.

Note: If installing on Windows, one of SiPANN's dependencies, `gdspy`, requires a C compiler for installation. This can be bypassed by first installing the `gdspy` wheel. This is done by downloading the wheel from gdspy, navigating to the location of the wheel, and executing

```
pip install gds*.whl
```

After this simply install SiPANN using your desired method.

4.2 SCEE and Interconnect

The SCEE module in SiPANN also has built in functionality to export any of it's models directly into a format readable by Lumerical Interconnect via the `export_interconnect()` function. This gives the user multiple options (Interconnect or Simphony) to cascade devices into complex structures. To export to a Interconnect file is as simple as a function call. First we declare all of our imports:

```
import numpy as np
from SiPANN import scee
```

Then make our device and calculate it's scattering parameters (we arbitrarily choose a half ring resonator here)

```
r = 10000
w = 500
t = 220
wavelength = np.linspace(1500, 1600)
gap = 100

hr = scee.HalfRing(w, t, r, gap)
sparams = hr.sparams(wavelength)
```

And then export. Note `export_interconnect` takes in wavelengths in nms, but the Lumerical file will have frequency in meters, as is standard in Interconnect. To export:

```
filename = "halfring_10microns_sparams.txt"
scee.export_interconnect(sparams, wavelength, filename)
```

As a final parameter, `export_interconnect` also has a `clear=True` parameter that will empty the file being written to before writing. If you'd like to append to an existing file, simply set `clear=False`.

This is available as a jupyter notebook here

4.3 Half Ring Resonator

We'll briefly show you how to simulate a half ring resonator using `SiPANN.scee`.

First do the imports

```
import numpy as np
import matplotlib.pyplot as plt
from SiPANN.scee import HalfRing

def pltAttr(x, y, title=None, legend='upper right', save=None):
    if legend is not None:
        plt.legend(loc=legend)
    plt.xlabel(x)
    plt.ylabel(y)
    if title is not None:
        plt.title(title)
    if save is not None:
        plt.savefig(save)
```

Declare all of our geometries (note they're all in nm)

```
r = 10000
w = 500
t = 220
wavelength = np.linspace(1500, 1600)
gap = 100
```

And we can simulate using either `Halfring.sparams` or `Halfring.predict()`. Using `predict()` this looks like

```
hr = HalfRing(w, t, r, gap)
k = hr.predict((1,4), wavelength)
t = hr.predict((1,3), wavelength)
```

And if you want to visualize what the device looks like,

```
hr.gds(view=True,extra=0,units='microns')
```

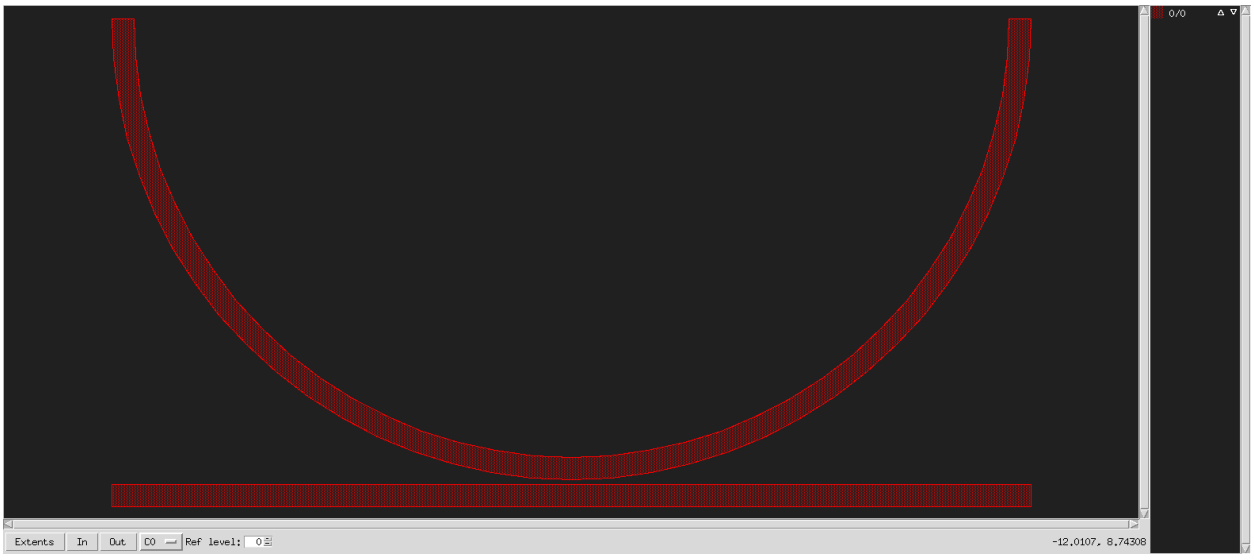
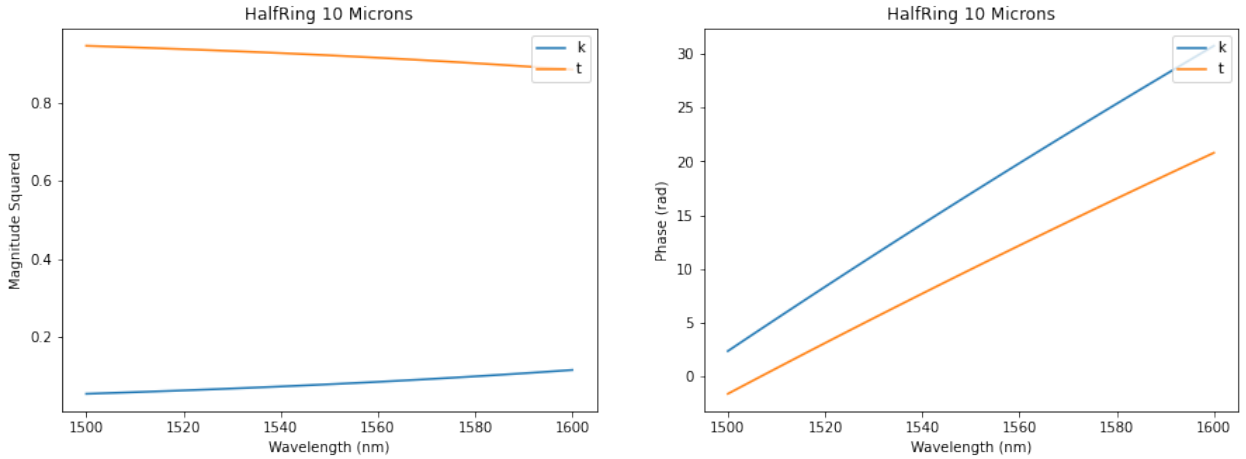


Fig. 1: halfring

And plotting the output gives us

```
plt.figure(figsize=(15,5))
plt.subplot(121)
plt.plot(wavelength, np.abs(k)**2, label='k')
plt.plot(wavelength, np.abs(t)**2, label='t')
pltAttr('Wavelength (nm)', 'Magnitude Squared', 'HalfRing 10 Microns')
plt.subplot(122)
plt.plot(wavelength, np.unwrap(np.angle(k)), label='k')
plt.plot(wavelength, np.unwrap(np.angle(t)), label='t')
pltAttr('Wavelength (nm)', 'Phase (rad)', 'HalfRing 10 Microns')
```

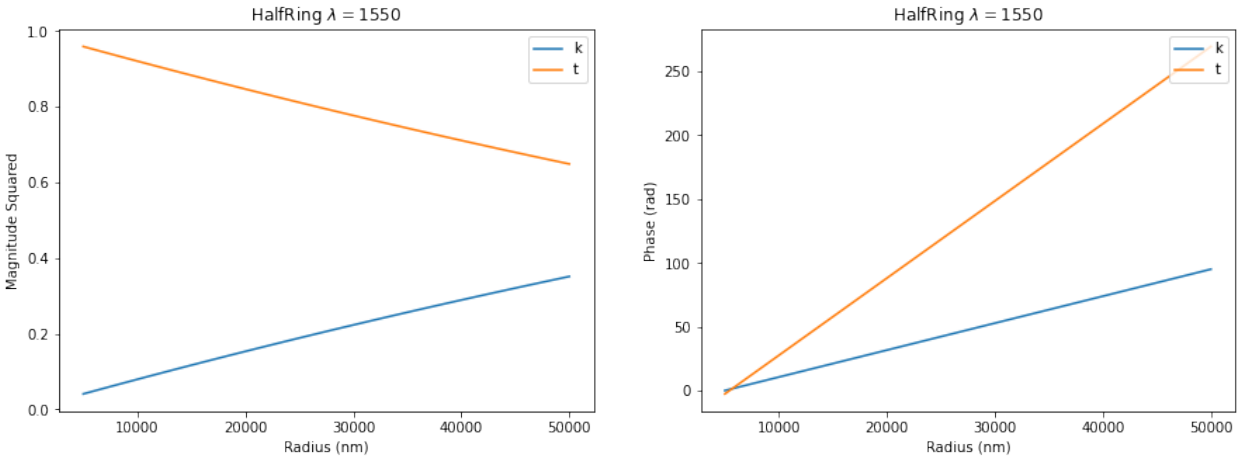


SCEE also supports sweeping over geometries while holding wavelengths fixed. This can be done as:

```
r = np.linspace(5000, 50000, 100)
wavelength = 1550
hr.update(radius=r)

k = hr.predict((1,4), wavelength)
t = hr.predict((1,3), wavelength)

plt.figure(figsize=(15,5))
plt.subplot(121)
plt.plot(r, np.abs(k)**2, label='k')
plt.plot(r, np.abs(t)**2, label='t')
pltAttr('Radius (nm)', 'Magnitude Squared', 'HalfRing $\lambda=1550$')
plt.subplot(122)
plt.plot(r, np.unwrap(np.angle(k)), label='k')
plt.plot(r, -np.unwrap(np.angle(t)), label='t')
pltAttr('Radius (nm)', 'Phase (rad)', 'HalfRing $\lambda=1550$')
```



All elements found in SiPANN.scee can be simulated basically identically. If you'd like this tutorial as a jupyter notebook, it can be found on github, here

4.4 Inverse Design using SCEE

Do to how fast SCEE is, inverse design of power splitting directional couplers can be achieved via an optimizer. This has been implemented and can be used via the `SiPANN.scee_opt` module, speficially the `make_coupler` function. It implements a global optimization, then a local optimization to best find the ideal coupler.

This is done by defining the length of the coupler and various control points along the coupler as parameters that our optimizer can choose that result in a κ closest to κ_{goal}. The coupler is then defined using the control points plugged into a Bezier Curve. Note that the Bezier curve defined by the control points is the gap between waveguides, not the geometry of the waveguides themselves. However, since each of these directional couplers is symmetric the inner boundary of the waveguides are just half of the gap.

Further, for our objective function, we compute κ for a sweep of wavelength points using SCEE, and then calculate the MSE by comparing it to κ_{goal}. Various constraints are also put on the coupler, like ensuring the edges of the coupler are far enough apart and making sure there's no bends that are too sharp. To learn more about the process, see [INSERT PAPER WHEN PUBLISHED].

```
import numpy as np
import matplotlib.pyplot as plt
from SiPANN import scee_opt, scee

def pltAttr(x, y, title=None, legend='upper right', save=None):
    if legend is not None:
        plt.legend(loc=legend)
    plt.xlabel(x)
    plt.ylabel(y)
    if title is not None:
        plt.title(title)
    if save is not None:
        plt.savefig(save)
```

4.4.1 Wavelength Sweep

Most of the defaults parameters for `SiPANN.scee_opt.make_coupler` have been set because they work well, so if you don't know what you're doing, it's generally best to stick with them. Here we make a coupler with 25%

of the light coming out of the cross port. It will return an instance of `SiPANN.scee.GapFuncSymmetric`, the control points chosen, and the final length of the coupler.

The number of points passed into `waveSweep` is the one that affects how long the optimizer takes the most. We pass in 2 points. This will cause the optimizer to use the endpoints of our wavelength sweep when evaluating the objective function. This is generally sufficient since the response is usually either increasing or decreasing everywhere, so trying to push the endpoints to our desired ratio is enough.

```
width = 500
thickness = 220
coupler, gap_pts, length = scee_opt.make_coupler(goalK=.25, verbose=1,
                                                 width=width, thickness=thickness,
                                                 waveSweep=np.array([1500,1600]),␣
↪maxiter=15000)
```

```
LOCAL, MSE: 0.0267, Mean currK: 0.2864:  50%|     | 15001/30000 [42:58<38:09,  6.55it/
↪s]
```

We can easily save it to an npz file for later use using `SiPANN.scee_opt.save_coupler`

```
scee_opt.save_coupler(width, thickness, gap_pts, length, "split_25_75.npz")
```

And then reload it using the `SiPANN.scee_opt.load_coupler`. This will return us an instance of `SiPANN.scee.GapFuncSymmetric`.

```
coupler, length = scee_opt.load_coupler("split_25_75.npz")
```

And we'll check what it looks like and make sure it performs as we anticipated.

```
coupler.gds(view=True,extra=0,units='microns')
```

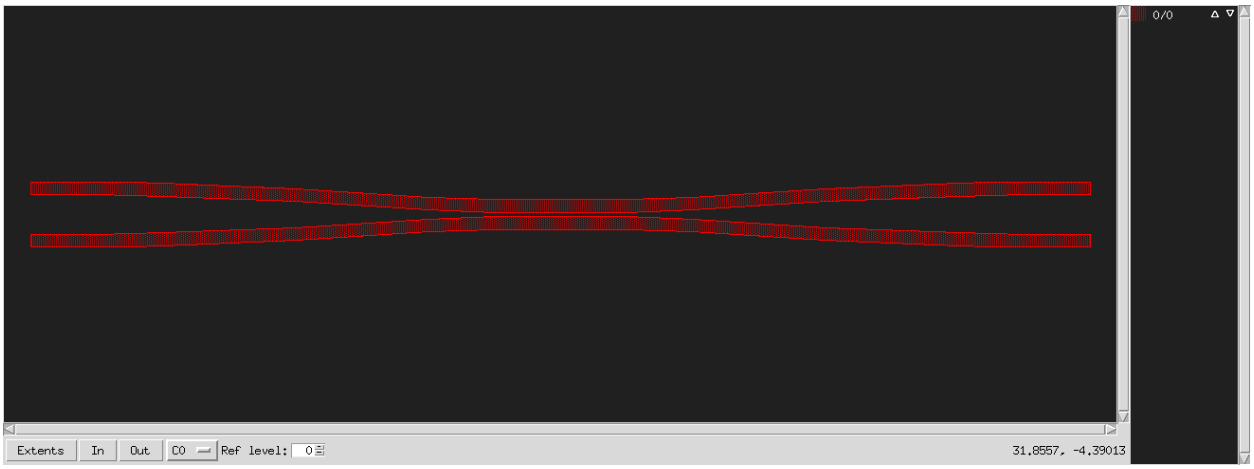


Fig. 2: coupler

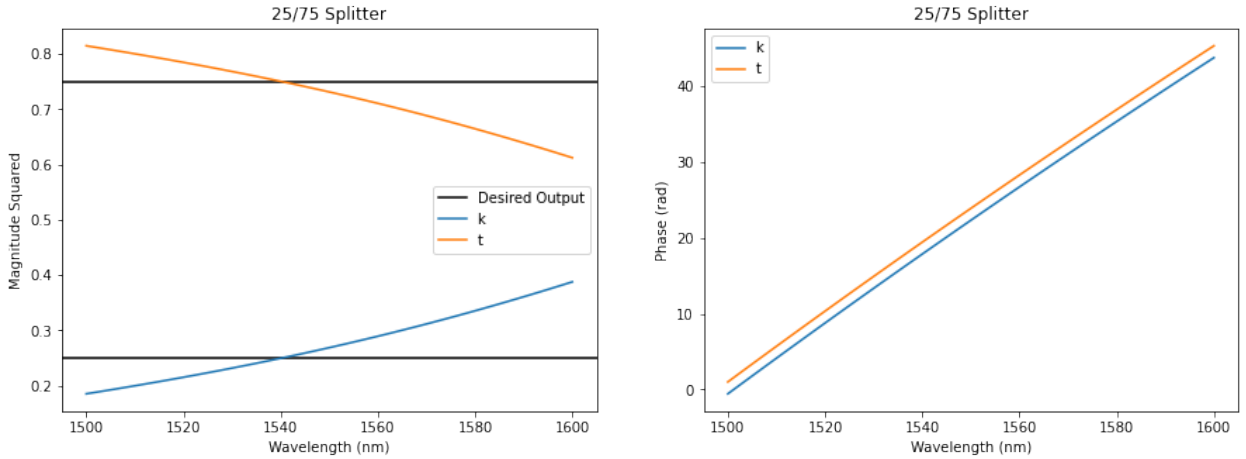
```
wavelength = np.linspace(1500, 1600, 100)
k = coupler.predict((1,4), wavelength)
t = coupler.predict((1,3), wavelength)

plt.figure(figsize=(15,5))
plt.subplot(121)
plt.axhline(.25, c='k', label="Desired Output")
```

(continues on next page)

(continued from previous page)

```
plt.axhline(.75, c='k')
plt.plot(wavelength, np.abs(k)**2, label='k')
plt.plot(wavelength, np.abs(t)**2, label='t')
pltAttr('Wavelength (nm)', 'Magnitude Squared', '25/75 Splitter', legend='best')
plt.subplot(122)
plt.plot(wavelength, np.unwrap(np.angle(k)), label='k')
plt.plot(wavelength, np.unwrap(np.angle(t)), label='t')
pltAttr('Wavelength (nm)', 'Phase (rad)', '25/75 Splitter', legend='best')
```



For your convenience, this has been done for a variety of splitting ratios already and they all can be loaded using `SiPANN.scee_opt.premade_coupler`. To learn more about how to use that see the tutorial on Premade Couplers via Inverse Design.

4.4.2 Single Wavelength

If we desire a specific wavelength to have a ratio, we can achieve this by making `waveSweep` only have one element. This causes only a single wavelength to be optimized (and a solution is generally found very quickly). This enforces nothing on any other wavelengths, so it's possible that it will be significantly different than the desired ratio even at close wavelengths.

For an example, we show this for $\lambda = 1530nm$ and a ratio of 45%.

```
width = 500
thickness = 220
coupler, gap_pts, length = scee_opt.make_coupler(goalK=.45, verbose=1,
                                                 width=width, thickness=thickness,␣
↪maxiter=30000,
                                                 waveSweep=np.array([1530]))
scee_opt.save_coupler(width, thickness, gap_pts, length, "split_45_55_1530.npz")
```

```
LOCAL, MSE: 0.1523, Mean currK: 0.1832:  18%|        | 10975/60000 [18:09<1:31:59,  8.
↪88it/s]
```

```
coupler, length = scee_opt.load_coupler("split_45_55_1530.npz")
coupler.gds(view=True,extra=0,units='microns')
```

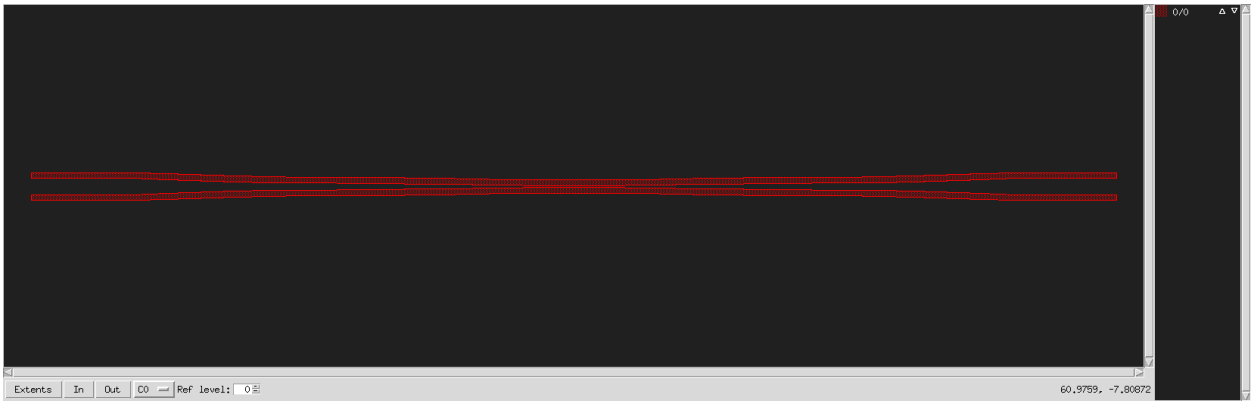
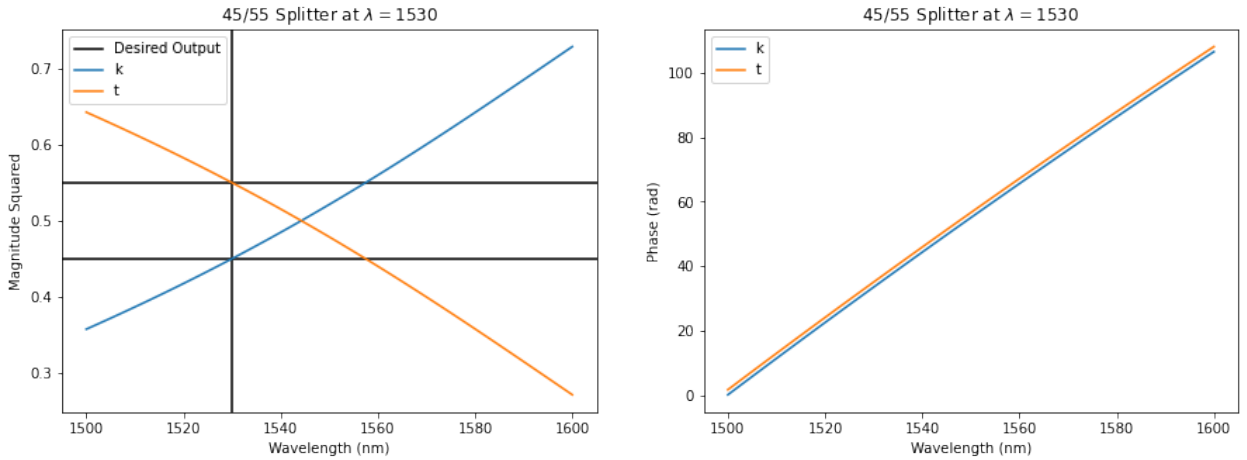


Fig. 3: splitter

```
wavelength = np.linspace(1500, 1600, 100)
k = coupler.predict((1,4), wavelength)
t = coupler.predict((1,3), wavelength)

plt.figure(figsize=(15,5))
plt.subplot(121)
plt.axhline(.45, c='k', label="Desired Output")
plt.axhline(.55, c='k')
plt.axvline(1530, c='k')
plt.plot(wavelength, np.abs(k)**2, label='k')
plt.plot(wavelength, np.abs(t)**2, label='t')
pltAttr('Wavelength (nm)', 'Magnitude Squared', '45/55 Splitter at $\lambda=1530$',
↪legend='best')
plt.subplot(122)
plt.plot(wavelength, np.unwrap(np.angle(k)), label='k')
plt.plot(wavelength, np.unwrap(np.angle(t)), label='t')
pltAttr('Wavelength (nm)', 'Phase (rad)', '45/55 Splitter at $\lambda=1530$', legend=
↪'best')
```



If you'd like this tutorial as a jupyter notebook, it can be found on github, here

4.5 Premade Couplers via Inverse Design

Using an inverse design optimizer and SCEE, various power splitting couplers at various splitting ratios have been designed and saved for future use. These can be loaded using `SiPANN.scee_opt.premade_coupler` module. We'll go through how to load them here.

```
import numpy as np
import matplotlib.pyplot as plt
from SiPANN import scee_opt, scee

def pltAttr(x, y, title=None, legend='upper right', save=None):
    if legend is not None:
        plt.legend(loc=legend)
    plt.xlabel(x)
    plt.ylabel(y)
    if title is not None:
        plt.title(title)
    if save is not None:
        plt.savefig(save)
```

4.5.1 Crossover

Premade couplers can be loaded via the `SiPANN.scee_opt.premade_coupler` function. It takes in a desired percentage output of the throughport. The only percentages available are 10, 20, 30, 40, 50, and 100 (crossover). It returns a instance of `SiPANN.scee.GapFuncSymmetric` with all it's usual functions and abilities, along with the coupler length in nanometers.

If you desire other ratios, see the tutorial on `SiPANN.scee_opt.make_coupler`, where the inverse design optimizer can be used to make arbitrary splitting ratios.

```
crossover, length = scee_opt.premade_coupler(100)
crossover.gds(view=True,extra=0,units='microns')
```

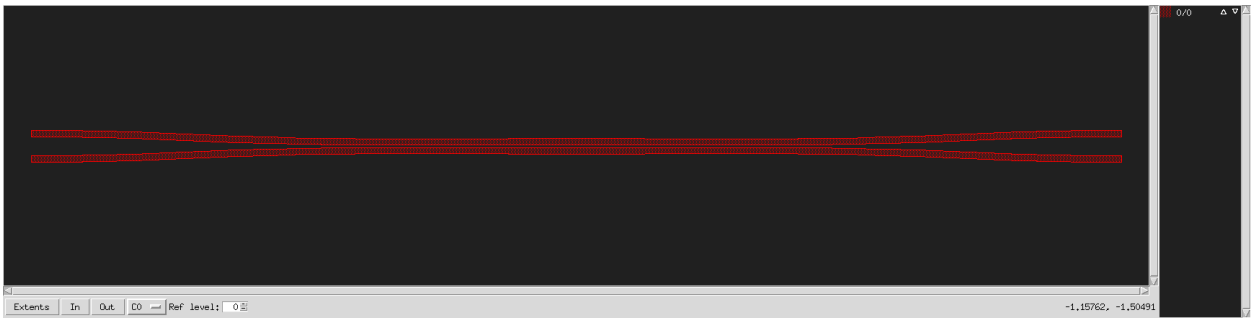


Fig. 4: crossover

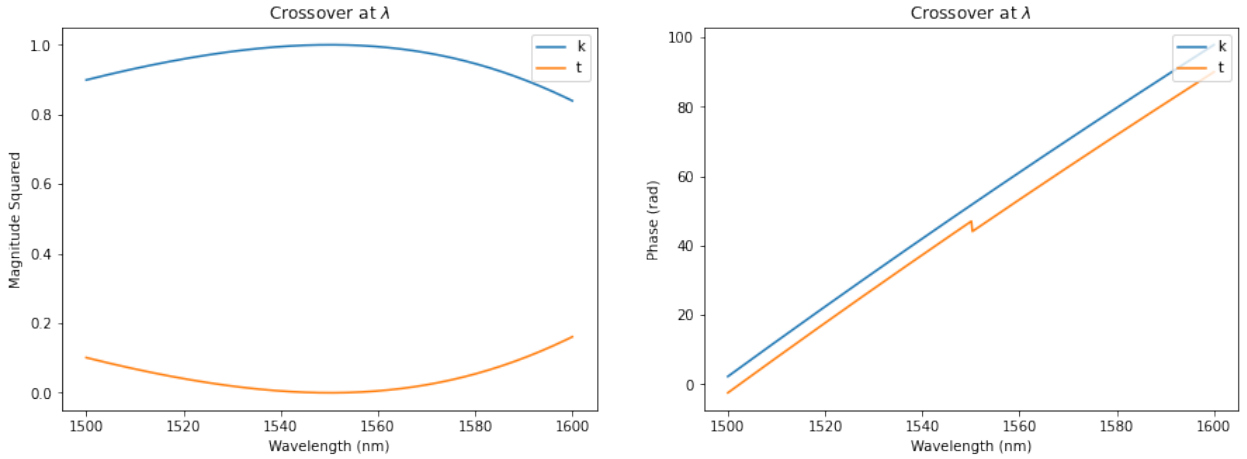
```
wavelength = np.linspace(1500, 1600, 500)
k = crossover.predict((1,4), wavelength)
t = crossover.predict((1,3), wavelength)

plt.figure(figsize=(15,5))
plt.subplot(121)
plt.plot(wavelength, np.abs(k)**2, label='k')
```

(continues on next page)

(continued from previous page)

```
plt.plot(wavelength, np.abs(t)**2, label='t')
pltAttr('Wavelength (nm)', 'Magnitude Squared', 'Crossover at $\lambda \approx 1550nm$
↪')
plt.subplot(122)
plt.plot(wavelength, np.unwrap(np.angle(k)), label='k')
plt.plot(wavelength, np.unwrap(np.angle(t)), label='t')
pltAttr('Wavelength (nm)', 'Phase (rad)', 'Crossover at $\lambda \approx 1550nm$')
```



4.5.2 30/70 Splitter

For further demonstration, we also load a 30/70 splitter.

```
splitter, length = scee_opt.premade_coupler(30)
splitter.gds(view=True,extra=0,units='microns')
```

```
wavelength = np.linspace(1500, 1600, 500)
k = splitter.predict((1,4), wavelength)
t = splitter.predict((1,3), wavelength)

plt.figure(figsize=(15,5))
plt.subplot(121)
plt.axhline(.3, c='k', label="Desired Ratios")
plt.axhline(.7, c='k')
plt.plot(wavelength, np.abs(k)**2, label='k')
plt.plot(wavelength, np.abs(t)**2, label='t')
pltAttr('Wavelength (nm)', 'Magnitude Squared', '30/70 Splitter', legend='center left
↪')
plt.subplot(122)
plt.plot(wavelength, np.unwrap(np.angle(k)), label='k')
plt.plot(wavelength, np.unwrap(np.angle(t)), label='t')
pltAttr('Wavelength (nm)', 'Phase (rad)', '30/70 Splitter')
```

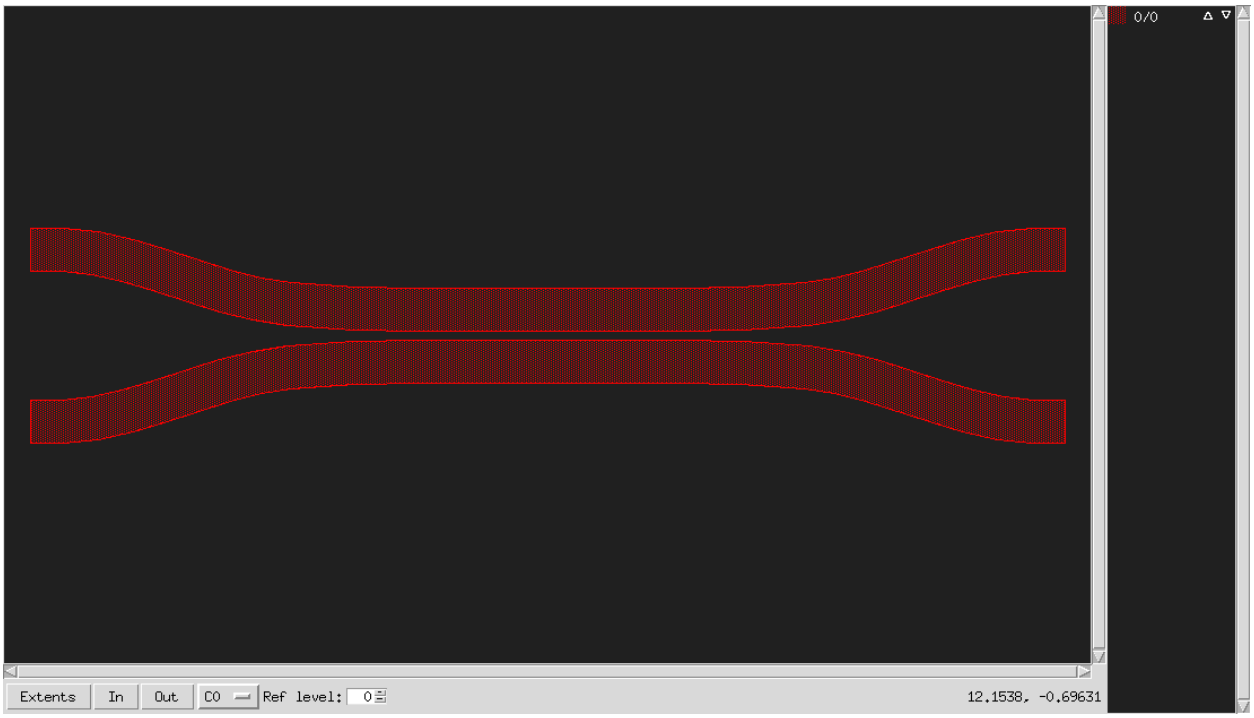
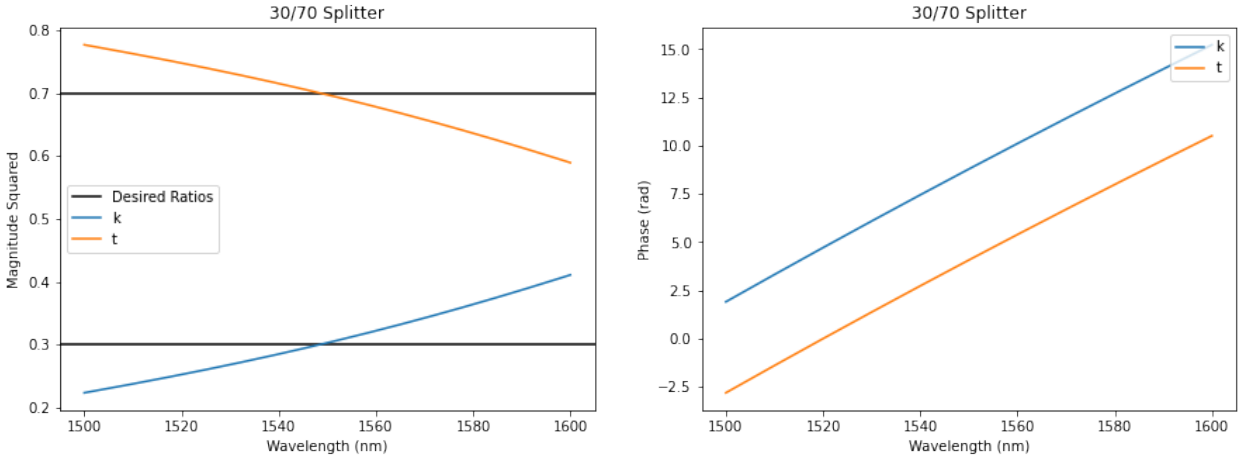


Fig. 5: splitter



If you'd like this tutorial as a jupyter notebook, it can be found on github, here

4.6 SCEE and Simphony

SiPANN includes a module that wraps all of the models produced by SCEE for easy use in simphony, allowing for cascading devices to make complex structures. This gives the user multiple options (Interconnect or Simphony) to cascade devices.

The SCEE wrapper can be found in `SiPANN.scee_int`.

```
from SiPANN import scee
from SiPANN.scee_int import SimphonyWrapper

from simphony.library import ebeam
from simphony.netlist import Subcircuit
from simphony.simulation import SweepSimulation, MonteCarloSweepSimulation

import matplotlib.pyplot as plt
import numpy as np

def pltAttr(x, y, title=None, legend='upper right', save=None):
    if legend is not None:
        plt.legend(loc=legend)
    plt.xlabel(x)
    plt.ylabel(y)
    if title is not None:
        plt.title(title)
    if save is not None:
        plt.savefig(save)
```

4.6.1 Standard Simulation

First we'll make our device like we always have using `SiPANN.scee`.

```
r = 10000
w = 500
t = 220
wavelength = np.linspace(1500, 1600)
gap = 100

hr = scee.HalfRing(w, t, r, gap)
```

Simply put our device into the simphony wrapper.

```
s_hr = SimphonyWrapper(hr)
```

Then use in simphony like you would any other device. Here we'll make a ring resonator as an example.

```
def make_ring(half_ring):
    term = ebeam.ebeam_terminator_te1550()

    circuit = Subcircuit()
    circuit.add([
        (half_ring, 'input'),
        (half_ring, 'output'),
        (term, 'terminator')
    ])

    circuit.elements['input'].pins = ('pass', 'midb', 'in', 'midt')
    circuit.elements['output'].pins = ('out', 'midt', 'term', 'midb')

    circuit.connect_many([
        ('input', 'midb', 'output', 'midb'),
        ('input', 'midt', 'output', 'midt'),
        ('terminator', 'n1', 'output', 'term')
    ])
```

(continues on next page)

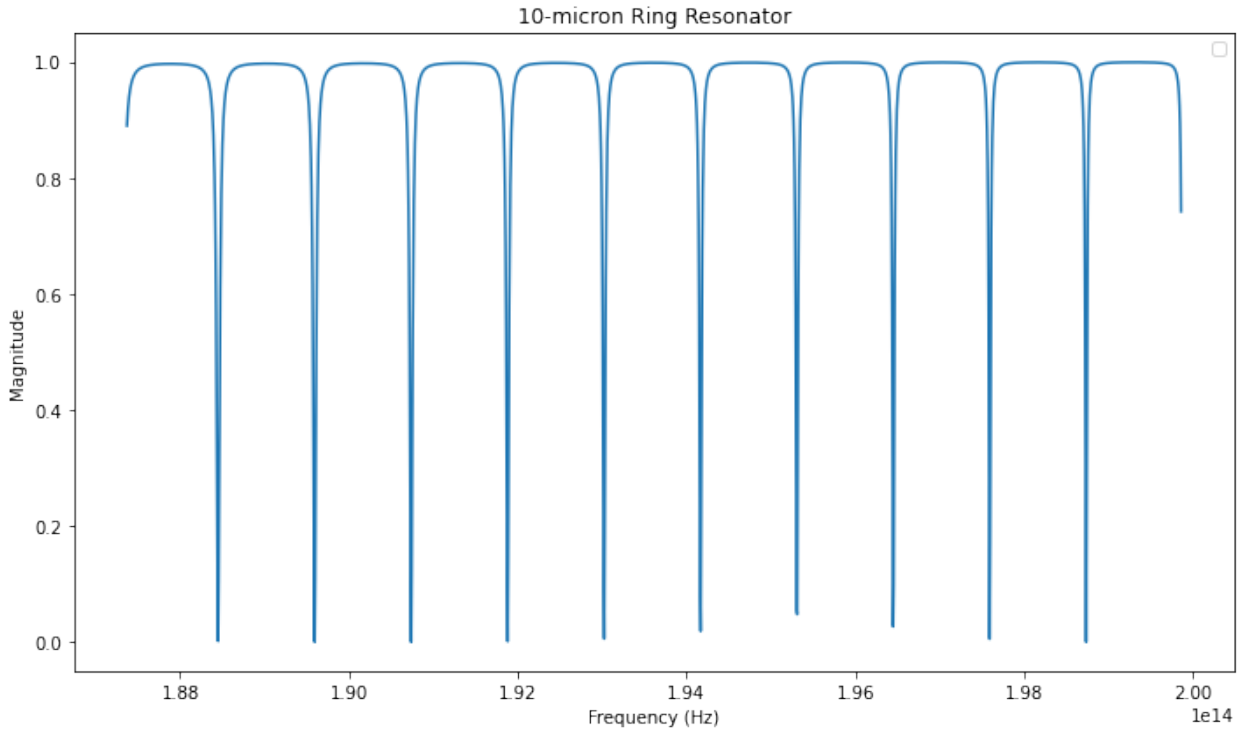
(continued from previous page)

```
    return circuit
```

```
# Simphony takes in wavelength values in meters
circuit = make_ring(s_hr)
sim1 = SweepSimulation(circuit, 1500e-9, 1600e-9)
res1 = sim1.simulate()

f1, s = res1.data(res1.pinlist['in'], res1.pinlist['pass'])
plt.figure(figsize=(10,6))
plt.plot(f1, s)
pltAttr('Frequency (Hz)', 'Magnitude', "10-micron Ring Resonator")
plt.tight_layout()
plt.show()
```

```
No handles with labels found to put in legend.
```



4.6.2 Monte-Carlo Simulations

`SimphonyWrapper` also functions with monte_carlo simulations. It allows ANY of the parameters set in SiPANN to be used. To use it you must include a dictionary mapping the parameter you wish to perturb to a standard deviation in nm.

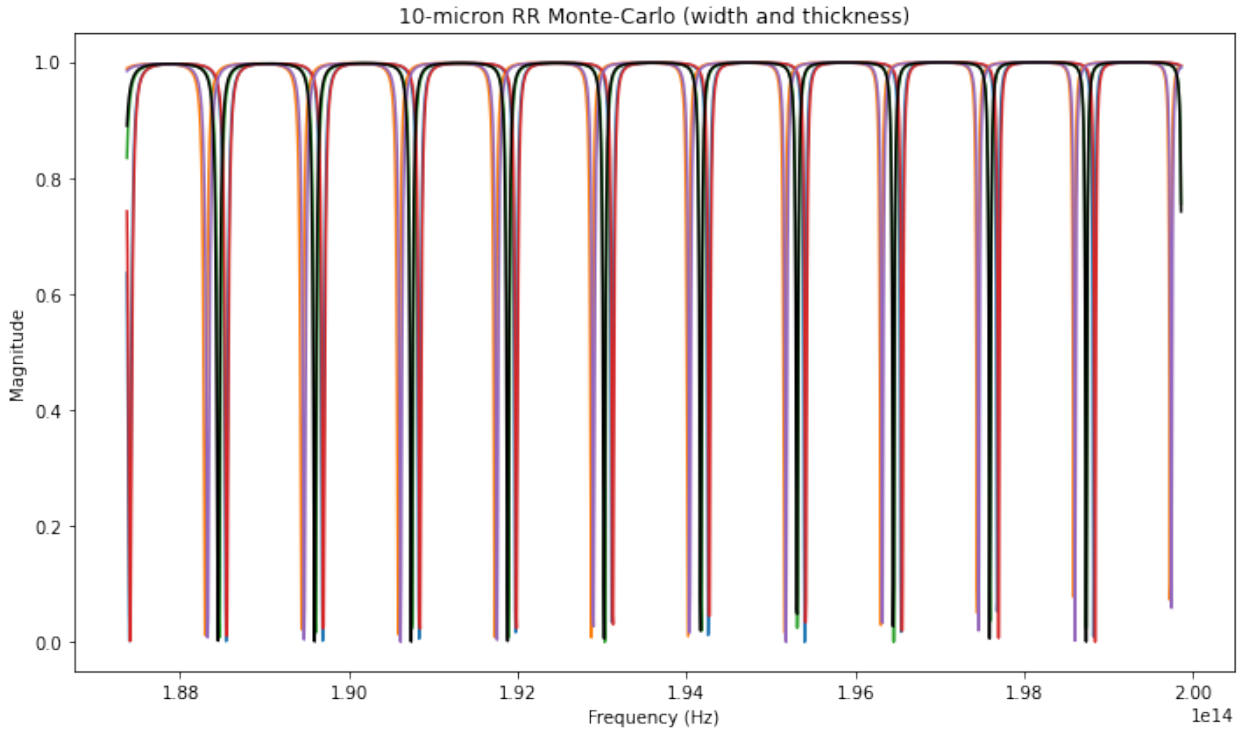
```
sigmas = {"width": 2, "thickness": 1}
s_hr = SimphonyWrapper(hr, sigmas)
```

And then simply make our circuit as before, and run through monte-carlo simulations

```
circuit = make_ring(s_hr)
#run monte carlo simulation
simulation = MonteCarloSweepSimulation(circuit, 1500e-9, 1600e-9)
runs = 5
result = simulation.simulate(runs=runs)

#plot
plt.figure(figsize=(10,6))
for i in range(1, runs + 1):
    f, s = result.data('in', 'pass', i)
    plt.plot(f, s)

# The data located at the 0 position is the ideal values.
f, s = result.data('in', 'pass', 0)
plt.plot(f, s, 'k')
pltAttr('Frequency (Hz)', 'Magnitude', "10-micron RR Monte-Carlo (width and thickness)
↪", legend=None)
plt.tight_layout()
plt.show()
```



As an example, we'll do another simulation, but this time varying radius of the ring only. Note we could vary both sides of the ring independently as well (ie the gap distance on each side isn't necessarily going to be equal), but for simplicity using our `make_ring` function we have identical halves.

```
sigmas = {"radius": 20}
s_hr = SimphonyWrapper(hr, sigmas)

circuit = make_ring(s_hr)
#run monte carlo simulation
simulation = MonteCarloSweepSimulation(circuit, 1500e-9, 1600e-9)
runs = 5
```

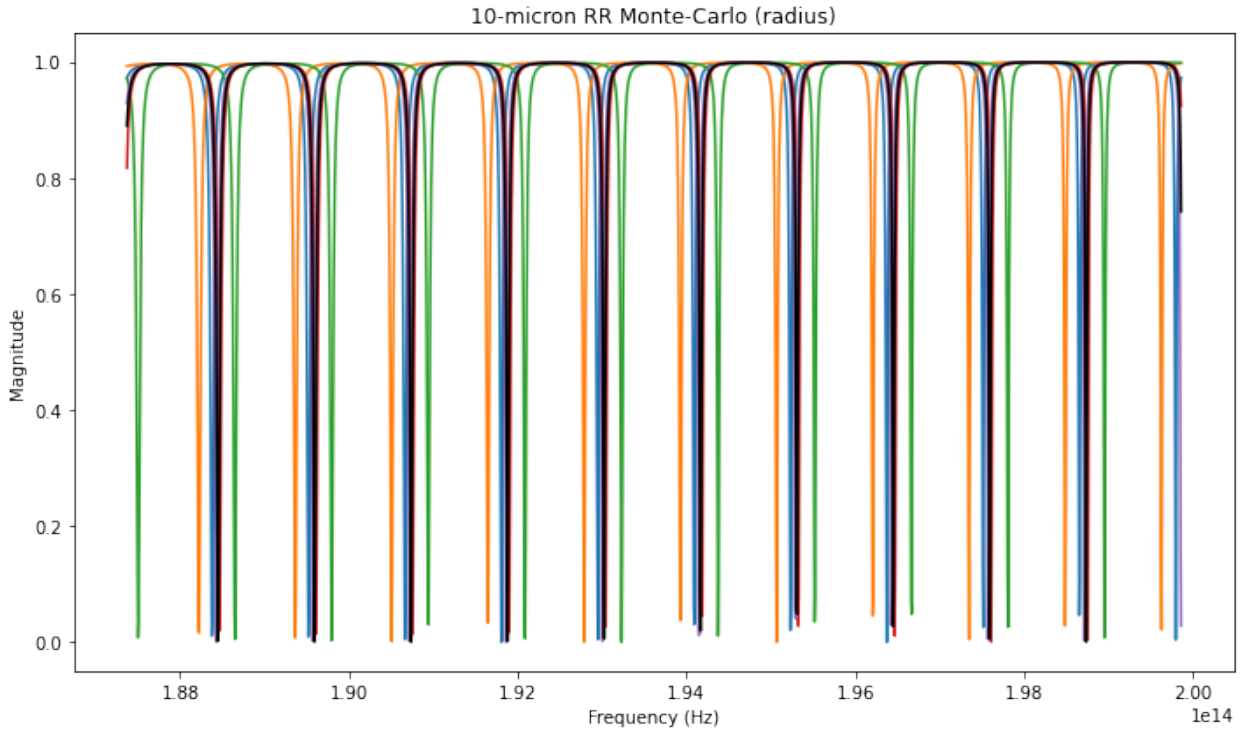
(continues on next page)

(continued from previous page)

```
result = simulation.simulate(runs=runs)

#plot
plt.figure(figsize=(10,6))
for i in range(1, runs + 1):
    f, s = result.data('in', 'pass', i)
    plt.plot(f, s)

# The data located at the 0 position is the ideal values.
f, s = result.data('in', 'pass', 0)
plt.plot(f, s, 'k')
pltAttr('Frequency (Hz)', 'Magnitude', "10-micron RR Monte-Carlo (radius)",␣
↪legend=None)
plt.tight_layout()
plt.show()
```



This is available as a jupyter notebook here

4.7 Composite Devices Models

These are models that combine the model used in SCEE along with various Neural Networks to make a full compact model.

4.7.1 Racetrack Resonator

class SiPANN.comp.**racetrack_sb_rr**(*width, thickness, radius, gap, length, sw_angle=90, loss=[0.99]*)

Racetrack waveguide arc used to connect to a racetrack directional coupler. Ports labeled as:

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm
- **radius** (*float or ndarray*) – Distance from center of ring to middle of waveguide in nm.
- **gap** (*float or ndarray*) – Minimum distance from ring waveguide edge to straight waveguide edge in nm.
- **length** (*float or ndarray*) – Length of straight portion of ring waveguide in nm.
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees. Defaults to 90.

update (***kwargs*)
Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict (*wavelength*)
Predicts the output when light is put in port 1 and out port 2.

Parameters **wavelength** (*float or ndarray*) – Wavelength(s) to predict at

Returns **k/t** – The value of the light coming through

Return type complex ndarray

sparams (*wavelength*)
Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at

Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (2,2,n) complex matrix of scattering parameters

gds (*filename=None*, *view=False*, *extra=0*, *units='nms'*)
Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.
- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.

4.8 Neural Network Based Models

These are various applications of Neural Networks and Linear Regressions we've trained to estimate effective indices.

4.8.1 Straight Waveguide Model

Straight Waveguide Effective Index

SiPANN.nn.**straightWaveguide**(*wavelength*, *width*, *thickness*, *sw_angle=90*, *derivative=None*)

Calculates the first effective index value of the TE mode. Can also calculate derivatives with respect to any of the inputs. This is especially useful for calculating the group index, or running gradient based optimization routines.

Each of the inputs can either be a one dimensional numpy array or a scalar. This is especially useful if you want to sweep over multiple parameters and include all of the possible permutations of the sweeps.

The output is a multidimensional array. The size of each dimension corresponds with the size of each of the inputs, such that DIM1 = size(wavelength) DIM2 = size(width) DIM3 = size(thickness) DIM4 = size(sw_angle)

So if I swept 100 wavelength points, 1 width, 10 possible thicknesses, and 2 sw_angles, then the dimension of each output effective index (or higher order derivative) would be: (100,1,10,2).

Parameters

- **wavelength** (*float or ndarray (W1,)*) – Wavelength points to evaluate
- **width** (*float or ndarray (W2,)*) – Width of the waveguides in microns
- **thickness** (*float or ndarray (T,)*) – Thickness of the waveguides in microns
- **sw_angle** (*float or ndarray (A,)*) – Sidewall angle from horizontal in degrees, ie 90 makes a square. Defaults to 90.
- **derivative** (*int*) – Order of the derivative to take. Defaults to None.

Returns **TE0** – First TE effective index with size (W1,W2,T,A,), or if derivative's are included (W1,W2,T,A,4,)

Return type ndarray

Straight Waveguide Scattering Parameters

SiPANN.nn.**straightWaveguide_S**(*wavelength*, *width*, *thickness*, *length*, *sw_angle=90*)

Calculates the analytic scattering matrix of a simple straight waveguide with length L.

Parameters

- **wavelength** (*ndarray (N,)*) – Wavelength points to evaluate
- **width** (*float*) – Width of the waveguides in microns
- **thickness** (*float*) – Thickness of the waveguides in microns
- **sw_angle** (*float*) – Sidewall angle from horizontal in degrees, ie 90 makes a square. Defaults to 90.
- **L** (*float*) – Length of the waveguide in microns

Returns **S** – Scattering matrix for each wavelength

Return type ndarray (N,2,2)

4.8.2 Bent Waveguide Model

Bent Waveguide Effective Index

SiPANN.nn.**bentWaveguide**(*wavelength*, *width*, *thickness*, *radius*, *sw_angle=90*, *derivative=None*)

Calculates the first effective index value of the TE mode of a bent waveguide. Can also calculate derivatives with respect to any of the inputs. This is especially useful for calculating the group index, or running gradient based optimization routines.

Each of the inputs can either be a one dimensional numpy array or a scalar. This is especially useful if you want to sweep over multiple parameters and include all of the possible permutations of the sweeps.

The output is a multidimensional array. The size of each dimension corresponds with the size of each of the inputs, such that DIM1 = size(wavelength) DIM2 = size(width) DIM3 = size(thickness) DIM4 = size(radius) DIM5 = size(sw_angle)

So if I swept 100 wavelength points, 1 width, 10 possible thicknesses, 5 radii, and 2 sw_angles, then the dimension of each output effective index (or higher order derivative) would be: (100,1,10,5,2).

Parameters

- **wavelength** (*float or ndarray (W1,)*) – Wavelength points to evaluate
- **width** (*float or ndarray (W2,)*) – Width of the waveguides in microns
- **thickness** (*float or ndarray (T,)*) – Thickness of the waveguides in microns
- **radius** (*float or ndarray (R,)*) – Radius of waveguide in microns.
- **sw_angle** (*float or ndarray (A,)*) – Sidewall angle from horizontal in degrees, ie 90 makes a square. Defaults to 90.
- **derivative** (*int*) – Order of the derivative to take. Defaults to None.

Returns **TE0** – First TE effective index with size (W1,W2,T,R,A,), or if derivative's are included (W1,W2,T,R,A,5,)

Return type ndarray

Bent Waveguide Scattering Parameters

SiPANN.nn.**bentWaveguide_S**(*wavelength*, *width*, *thickness*, *radius*, *angle*, *sw_angle=90*)

Calculates the analytic scattering matrix of bent waveguide with specific radius and circle length.

Parameters

- **wavelength** (*ndarray (N,)*) – Wavelength points to evaluate
- **width** (*float*) – Width of the waveguides in microns
- **thickness** (*float*) – Thickness of the waveguides in microns
- **radius** (*float*) – Radius of waveguide in microns.
- **angle** (*float*) – Number of radians of circle that bent waveguide transverses
- **sw_angle** (*float*) – Sidewall angle from horizontal in degrees, ie 90 makes a square. Defaults to 90.

Returns **S** – Scattering matrix for each wavelength

Return type ndarray (N,2,2)

4.8.3 Evanescent Waveguide Coupler Model

Evanescent Waveguide Coupler Effective Indices

SiPANN.nn.**evWGcoupler**(*wavelength*, *width*, *thickness*, *gap*, *sw_angle=90*, *derivative=None*)

Calculates the even and odd effective indice values of the TE mode of parallel waveguides. Can also calculate derivatives with respect to any of the inputs. This is especially useful for calculating the group index, or running gradient based optimization routines.

Each of the inputs can either be a one dimensional numpy array or a scalar. This is especially useful if you want to sweep over multiple parameters and include all of the possible permutations of the sweeps.

The output is a multidimensional array. The size of each dimension corresponds with the size of each of the inputs, such that DIM1 = size(wavelength) DIM2 = size(width) DIM3 = size(thickness) DIM4 = size(gap) DIM5 = size(sw_angle)

So if I swept 100 wavelength points, 1 width, 10 possible thicknesses, 5 radii, and 2 sw_angles, then the dimension of each output effective index (or higher order derivative) would be: (100,1,10,5,2).

Parameters

- **wavelength** (*float or ndarray (W1,)*) – Wavelength points to evaluate
- **width** (*float or ndarray (W2,)*) – Width of the waveguides in microns
- **thickness** (*float or ndarray (T,)*) – Thickness of the waveguides in microns
- **gap** (*float or ndarray (G,)*) – Gap distance between waveguides
- **sw_angle** (*float or ndarray (A,)*) – Sidewall angle from horizontal in degrees, ie 90 makes a square. Defaults to 90.
- **derivative** (*int*) – Order of the derivative to take. Defaults to None.

Returns **TE0** – First TE effective index with size (W1,W2,T,G,A,), or if derivative's are included (W1,W2,T,G,A,5,)

Return type ndarray

Evanescent Waveguide Coupler Scattering Parameters

SiPANN.nn.**evWGcoupler_S**(*wavelength*, *width*, *thickness*, *gap*, *couplerLength*, *sw_angle=90*)

Calculates the analytic scattering matrix of a simple, parallel waveguide directional coupler using the ANN.

Parameters

- **wavelength** (*ndarray (N,)*) – Wavelength points to evaluate
- **width** (*float*) – Width of the waveguides in microns
- **thickness** (*float*) – Thickness of the waveguides in microns
- **gap** (*float*) – gap in the coupler region in microns
- **couplerLength** (*float*) – Length of the coupling region in microns

Returns **S** – Scattering matrix

Return type ndarray (N,4,4)

4.8.4 Racetrack Resonator Models

Racetrack Resonator Model 1

SiPANN.nn.**racetrack_AP_RR**(*wavelength*, *radius=5*, *couplerLength=5*, *gap=0.2*, *width=0.5*, *thickness=0.2*)

This particular transfer function assumes that the coupling sides of the ring resonator are straight, and the other two sides are curved. Therefore, the roundtrip length of the RR is 2*pi*radius + 2*couplerLength.

We assume that the round parts of the ring have negligble coupling compared to the straight sections.

Parameters

- **wavelength** (*ndarray (N,)*) – Wavelength points to evaluate
- **radius** (*float*) – Radius of the sides in microns
- **couplerLength** (*float*) – Length of the coupling region in microns
- **gap** (*float*) – Gap in the coupler region in microns
- **width** (*float*) – Width of the waveguides in microns
- **thickness** (*float*) – Thickness of the waveguides in microns

Returns **S** – Scattering matrix

Return type ndarray (N,4,4)

Racetrack Resonator Model 2

SiPANN.nn.**racetrack_AP_RR_TF**(*wavelength, sw_angle=90, radius=12, couplerLength=4.5, gap=0.2, width=0.5, thickness=0.2, widthCoupler=0.5, loss=[0.99], coupling=[0]*)

This particular transfer function assumes that the coupling sides of the ring resonator are straight, and the other two sides are curved. Therefore, the roundtrip length of the RR is 2*pi*radius + 2*couplerLength. This model also includes loss. (??? Need Verification on last line)

We assume that the round parts of the ring have negligble coupling compared to the straight sections.

Parameters

- **wavelength** (*ndarray (N,)*) – Wavelength points to evaluate
- **radius** (*float*) – Radius of the sides in microns
- **couplerLength** (*float*) – Length of the coupling region in microns
- **gap** (*float*) – Gap in the coupler region in microns
- **width** (*float*) – Width of the waveguides in microns
- **thickness** (*float*) – Thickness of the waveguides in microns

Returns

- **E** (*ndarray*) – Complex array of size (N,)
- **alpha** (*ndarray*) – Array of size (N,)
- **t** (*ndarray*) – Array of size (N,)
- **alpha_s** (*ndarray*) – Array of size (N,)
- **phi** (*ndarray*) – Array of size (N,)

Racetrack Resonator Model 3

SiPANN.nn.**rectangularRR**(*wavelength*, *radius=5*, *couplerLength=5*, *sideLength=5*, *gap=0.2*, *width=0.5*, *thickness=0.2*)

This particular transfer function assumes that all four sides of the ring resonator are straight and that the corners are rounded. Therefore, the roundtrip length of the RR is 2*pi*radius + 2*couplerLength + 2*sideLength.

We assume that the round parts of the ring have negligble coupling compared to the straight sections.

Parameters

- **wavelength** (*ndarray (N,)*) – Wavelength points to evaluate
- **radius** (*float*) – Radius of the sides in microns
- **couplerLength** (*float*) – Length of the coupling region in microns
- **sideLength** (*float*) – Length of each side not coupling in microns
- **gap** (*float*) – Gap in the coupler region in microns
- **width** (*float*) – Width of the waveguides in microns
- **thickness** (*float*) – Thickness of the waveguides in microns

Returns **S** – Scattering matrix

Return type ndarray (N,4,4)

4.9 Neural Network Utilities

These are a couple of classes that help use perform the operations we need on our neural networks and linear regressions.

4.9.1 TensorMinMax (sklearn MinMaxScaler)

class SiPANN.import_nn.**TensorMinMax**(*feature_range=(0, 1)*, *copy=True*)

Copy of sklearn's MinMaxScaler implemented to work with tensorflow.

When used, tensorflow is able to take gradients on the transformation as well as on the network itself, allowing for gradient-based optimization in inverse design problems.

Parameters

- **feature_range** (*2-tuple, optional*) – Desired range of transformed data. Defaults to (0, 1)
- **copy** (*bool, optional*) – Set to false to perform inplace operations. Defaults to True.

fit(*X*)

Fits the transfomer to the data.

Essentially finds original min and max of data to be able to shift the data.

Parameters **X** (*tensor or ndarray*) – Data to fit

transform(*X*, *mode='numpy'*)

Actually does the transorm.

Parameters

- **X** (*tensor or ndarray*) – Data to transform

- **mode** (*{'numpy' or 'tensor'}, optional*) – Whether to use numpy or tensorflow operations.

Returns **X** – Transformed data

Return type tensor or ndarray

inverse_transform(*X*, *mode='numpy'*)
Undo the transorm.

Parameters

- **X** (*tensor or ndarray*) – Data to inverse transform
- **mode** (*{'numpy' or 'tensor'}, optional*) – Whether to use numpy or tensorflow operations.

Returns **X** – Inverse transformed data

Return type tensor or ndarray

4.9.2 Neural Network Importer

class SiPANN.import_nn.**ImportNN**(*directory*)
Class to import trained NN.

This the way we've been saving and using our neural networks. After saving them we can simply import them using this class and it keeps them open for as many operations as we desire.

normX
Norm of the inputs

Type *TensorMinMax*

normY
Norm of the outputs

Type *TensorMinMax*)

s_data
Dimensions (size) of input and outputs

Type 2-tuple

Parameters **directory** (*str*) – The directory where the model has been stored

validate_input(*input*)
Used to check for valid input.

If it is only a single data point, expands the dimensions so it fits properly

Parameters **input** (*ndarray*) – Numpy array with width s_data[0] (hopefully)

Returns **input** – Numpy array with width s_data[0] (hopefully) and height 1

Return type ndarray

output(*input*, *kp=1*)
Runs input through neural network.

Parameters

- **input** (*ndarray*) – Numpy array with width s_data[0]

- **kp** (*int, optional*) – Value from 0 to 1, 1 refers to not performing any dropout on nodes, 0 drops all of them. Defaults to 1.

Returns **output** – numpy array with width s_data[1]

Return type ndarray

differentiate(*input*, *d*, *kp=1*)

Returns partial derivative of neural network.

Parameters

- **input** (*ndarray*) – numpy array with width s_data[0]
- **d** (*3-tuple of ints*) – Refers to partial of first element wrt second element to the order of third element
- **kp** (*int, optional*) – Value from 0 to 1, 1 refers to not performing any dropout on nodes, 0 drops all of them. Defaults to 1.

Returns **output** – numpy array with width s_data[1]

Return type ndarray

rel_error(*input*, *output*, *kp=1*)

Returns relative error of network.

Parameters

- **input** (*ndarray*) – Numpy array with width s_data[0]
- **output** (*ndarray*) – Numpy array with width s_data[1]
- **kp** (*int, optional*) – Value from 0 to 1, 1 refers to not performing any dropout on nodes, 0 drops all of them. Defaults to 1.

Returns **relative error** – The relative error of inputs/outputs

Return type scalar

4.9.3 Linear Regression Importer

class SiPANN.import_nn.**ImportLR**(*directory*)

Class to import trained Linear Regression.

To remove independence on sklearn and it's updates, we manually implement an sklearn Pipeline that includes (PolynomialFeatures, LinearRegression). We use the actual sklearn implementation to train, save the coefficients, and then proceed to implement it here. To see how to save a pipeline like above to be used here see SiPANN/LR/regress.py

coef_

Linear Regression Coefficients

Type ndarray

degree_

Degree to be used in PolynomialFeatures.

Type float

s_data

Dimensions of inputs and outputs

Type 2-tuple

Parameters **directory** (*str*) – The directory where the model has been stored

make_combos(*X*)
Duplicates Polynomial Features.

Takes in an input X, and makes all possibly combinations of it using polynomials of specified degree.

Parameters **X** (*ndarray*) – Numpy array of size (N, s_data[0])

Returns **polyCombos** – Numpy array of size (N,)

Return type ndarray

validate_input(*input*)
Used to check for valid input.

If it is only a single data point, expands the dimensions so it fits properly

Parameters **input** (*ndarray*) – Numpy array with width s_data[0] (hopefully)

Returns **input** – Numpy array with width s_data[0] (hopefully) and height 1

Return type ndarray

predict(*X*)
Predict values.

Runs X through Pipeline to make prediction

Parameters **X** (*ndarray*) – Numpy array of size (N, s_data[0])

Returns **polyCombos** – Numpy array of size (N,)

Return type ndarray

4.10 SCEE - Directional Couplers Models

These methods are based on the model SCEE found in [CITE PAPER WHEN PUBLISHED]. Valid ranges for wavelength is 1450nm-1650nm, width 400nm-600nm, thickness 180nm-240nm, sidewall angle 80-90 degrees and gap distance greater than 100nm. Will issue warning when outside of these ranges since results will likely be inaccurate.

4.10.1 Helper Functions

Base Directional Coupler Class

class SiPANN.scee.**DC**(*width*, *thickness*, *sw_angle=90*)
Abstract Class that all directional couplers inherit from. Each DC will inherit from it.

Base Class for DC. All other DC classes should be based on this one, including same functions (so documentation should be the same/similar with exception of device specific attributes). Ports are numbered as:

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update (***kwargs*)
Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

sparams (*wavelength*)
Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

predict (*ports*, *wavelength*)
Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)
- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds (*filename=None*, *extra=0*, *units='microns'*, *view=False*, *sbend_h=0*, *sbend_v=0*)
Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.
- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.

- **sbend_h** (*int, optional*) – How high to horizontally make additional sbends to move ports farther away. Sbends insert after extra. Only available in couplers with all horizontal ports (units same as units parameters). Defaults to 0
- **sbend_v** (*int, optional*) – Same as sbend_h, but vertical distance. Defaults to 0.

Waveguide

class SiPANN.scee.**Waveguide**(*width*, *thickness*, *length*, *sw_angle=90*)

Lossless model for a straight waveguide.

Simple model that makes sparameters for a straight waveguide. May not be the best option, but plays nice with other models in SCEE. Ports are numbered as:

```
|  1 ----------- 2   |
```

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **length** (*float or ndarray*) – Length of waveguide in nm.
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update(***kwargs*)

Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

sparams(*wavelength*)

Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

Returns **s_matrix** – size (2,2,n) complex matrix of scattering parameters

Return type ndarray

predict(*wavelength*)

Predicts the output when light is put in port 1 and out port 2.

Parameters **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds(*filename=None*, *extra=0*, *units='microns'*, *view=False*)

Writes the geometry to the gds file.

Parameters

- **filename** (*str*) – location to save file to, or if you don't want to defaults to None
- **extra** (*int*) –

  extra straight portion to add to ends of waveguides to make room in simulation
  (input with units same as units input)
- **units** (*str*) – either 'microns' or 'nms'. Units to save gds file in

Effective Index Finder

SiPANN.scee.**get_neff**(*wavelength*, *width*, *thickness*, *sw_angle=90*)

Return neff for a given waveguide profile.

Leverages Multivariate Linear Regression that maps wavelength, width, thickness and sidewall angle to effective index with silicon core and silicon dioxide cladding

Parameters

- **wavelength** (*float or ndarray*) – wavelength (Valid for 1450nm-1650nm)
- **width** (*float or ndarray*) – width (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – thickness (Valid for 180nm-240nm)
- **sw_angle** (*float or ndarray*) – sw_angle (Valid for 80-90 degrees)

Returns **neff** – effective index of waveguide

Return type float or ndarray

Integrations Coefficient Finder

SiPANN.scee.**get_coeffs**(*wavelength*, *width*, *thickness*, *sw_angle*)

Return coefficients and neff for a given waveguide profile as used in SCEE.

Leverages Multivariate Linear Regression that maps wavelength, width, thickness and sidewall angle to effective index and coefficients used in estimate of even and odd effective indices with silicon core and silicon dioxide cladding.

Parameters

- **wavelength** (*float or ndarray*) – wavelength (Valid for 1450nm-1650nm)
- **width** (*float or ndarray*) – width (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – thickness (Valid for 180nm-240nm)
- **sw_angle** (*float or ndarray*) – sw_angle (Valid for 80-90 degrees)

Returns

- **ae** (*float or ndarray*) – used in even mode estimation in neff + ae exp(ge * g)
- **ao** (*float or ndarray*) – used in odd mode estimation in neff + ao exp(go * g)
- **ge** (*float or ndarray*) – used in even mode estimation in neff + ae exp(ge * g)
- **go** (*float or ndarray*) – used in odd mode estimation in neff + ao exp(go * g)
- **neff** (*float or ndarray*) – effective index of waveguide

Input Cleaner

SiPANN.scee.**clean_inputs**(*inputs*)

Makes all inputs as the same shape to allow passing arrays through.

Used to make sure all inputs have the same length - ie that it's trying to run a specific number of simulations, not a varying amount

Parameters **inputs** (*tuple*) – can be an arbitrary mixture of floats/ndarray

Returns **inputs** – returns all inputs as same size ndarrays

Return type tuple

4.10.2 Coupling Devices

Symmetric Coupler

class SiPANN.scee.**GapFuncSymmetric**(*width*, *thickness*, *gap*, *dgap*, *zmin*, *zmax*, *sw_angle=90*)

This class will create arbitrarily shaped SYMMETRIC (ie both waveguides are same shape) directional couplers.

It takes in a gap function that describes the gap as one progreses through the device. Note that the shape fo the waveguide will simply be half of gap function. Also requires the derivative of the gap function for this purpose. Ports are numbered as:

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **gap** (*function*) – Gap function as one progresses along the waveguide (Must always be > 100nm)
- **dgap** (*function*) – Derivative of the gap function
- **zmin** (*float*) – Where to begin integration in the gap function
- **zmax** (*float*) – Where to end integration in the gap function
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update(***kwargs*)

Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict(*ports*, *wavelength*, *extra_arc=0*, *part='both'*)

Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)
- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds (*filename=None*, *extra=0*, *units='microns'*, *view=False*, *sbend_h=0*, *sbend_v=0*)
Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.
- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.
- **sbend_h** (*int, optional*) – How high to horizontally make additional sbends to move ports farther away. Sbends insert after extra. Only available in couplers with all horizontal ports (units same as units parameters). Defaults to 0
- **sbend_v** (*int, optional*) – Same as sbend_h, but vertical distance. Defaults to 0.

sparams (*wavelength*)
Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

Non-Symmetric Coupler

class SiPANN.scee.**GapFuncAntiSymmetric**(*width*, *thickness*, *gap*, *zmin*, *zmax*, *arc1*, *arc2*, *arc3*, *arc4*, *sw_angle=90*)
This class will create arbitrarily shaped ANTISYMMETRIC (ie waveguides are different shapes) directional couplers.

It takes in a gap function that describes the gap as one progreses through the device. Also takes in arc length of each port up till coupling point. Ports are numbered as: | 2— /—4 | | —— | | —— | | 1—/ —3 |

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **gap** (*function*) – Gap function as one progresses along the waveguide (Must always be > 100nm)
- **zmin** (*float*) – Where to begin integration in the gap function
- **zmax** (*float*) – Where to end integration in the gap function
- **arc2, arc3, arc4** (*arc1,*) – Arclength from entrance of each port till minimum coupling point
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update (***kwargs*)

Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict (*ports*, *wavelength*, *extra_arc=0*, *part='both'*)

Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)
- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds (*filename=None*, *extra=0*, *units='microns'*, *view=False*, *sbend_h=0*, *sbend_v=0*)

Still needs to be implemented for this class.

sparams (*wavelength*)

Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

Coupling Straight Waveguides

class SiPANN.scee.**StraightCoupler**(*width*, *thickness*, *gap*, *length*, *sw_angle=90*)
This class will create half of a ring resonator.

It takes in a radius and gap along with usual waveguide parameters. Ports are numbered as:

```
|       2---------------4       |
|       1---------------3       |
```

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **gap** (*float or ndarray*) – Distance between the two waveguides edge in nm. (Must be > 100nm)
- **length** (*float or ndarray*) – Length of both waveguides in nm.
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update(***kwargs*)
Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict(*ports*, *wavelength*)
Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)
- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds(*filename=None*, *view=False*, *extra=0*, *units='nms'*, *sbend_h=0*, *sbend_v=0*)
Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.

- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.
- **sbend_h** (*int, optional*) – How high to horizontally make additional sbends to move ports farther away. Sbends insert after extra. Only available in couplers with all horizontal ports (units same as units parameters). Defaults to 0
- **sbend_v** (*int, optional*) – Same as sbend_h, but vertical distance. Defaults to 0.

sparams(*wavelength*)
Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

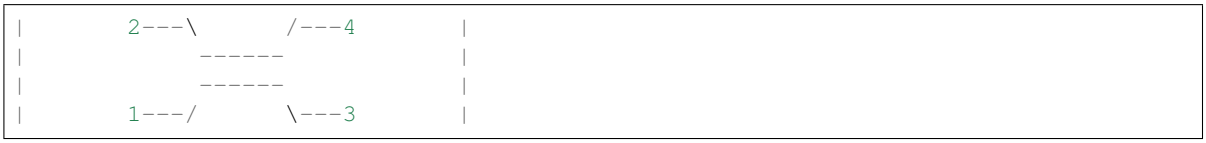
Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

Standard Directional Coupler

class SiPANN.scee.**Standard**(*width*, *thickness*, *gap*, *length*, *H*, *V*, *sw_angle=90*)
Normal/Standard Shaped Directional Coupler.

This is what most people think of when they think directional coupler. Ports are numbered as:



Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **gap** (*float or ndarray*) – Minimum distance between the two waveguides edge in nm. (Must be > 100nm)
- **length** (*float or ndarray*) – Length of the straight portion of both waveguides in nm.
- **H** (*float or ndarray*) – Horizontal distance between end of coupler until straight portion in nm.
- **H** – Vertical distance between end of coupler until straight portion in nm.
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update(***kwargs*)
Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict (*ports*, *wavelength*)
Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)
- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds (*filename=None*, *view=False*, *extra=0*, *units='nms'*, *sbend_h=0*, *sbend_v=0*)
Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.
- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.
- **sbend_h** (*int, optional*) – How high to horizontally make additional sbends to move ports farther away. Sbends insert after extra. Only available in couplers with all horizontal ports (units same as units parameters). Defaults to 0
- **sbend_v** (*int, optional*) – Same as sbend_h, but vertical distance. Defaults to 0.

sparams (*wavelength*)
Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

Half Ring

class SiPANN.scee.**HalfRing** (*width*, *thickness*, *radius*, *gap*, *sw_angle=90*)
This class will create half of a ring resonator.

It takes in a radius and gap along with usual waveguide parameters. Ports are numbered as:

```
|          2 \      / 4          |
|             \    /             |
|              ---               |
|          1----------3          |
```

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **radius** (*float or ndarray*) – Distance from center of ring to middle of waveguide in nm.
- **gap** (*float or ndarray*) – Minimum distance from ring waveguide edge to straight waveguide edge in nm. (Must be > 100nm)
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update (***kwargs*)
Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict (*ports*, *wavelength*)
Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)
- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds (*filename=None*, *view=False*, *extra=0*, *units='nms'*)
Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.
- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.

sparams (*wavelength*)
Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

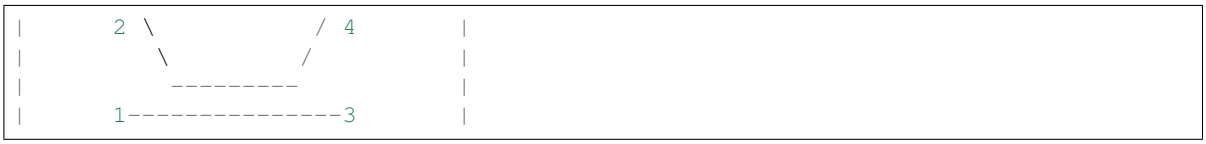
Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

Half Racetrack Resonator

class SiPANN.scee.**HalfRacetrack**(*width*, *thickness*, *radius*, *gap*, *length*, *sw_angle=90*)
This class will create half of a ring resonator.

It takes in a radius and gap along with usual waveguide parameters. Ports are numbered as:



Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **radius** (*float or ndarray*) – Distance from center of ring to middle of waveguide in nm.
- **gap** (*float or ndarray*) – Minimum distance from ring waveguide edge to straight waveguide edge in nm. (Must be > 100nm)
- **length** (*float or ndarray*) – Length of straight portion of ring waveguide in nm.
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update (***kwargs*)
Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict (*ports*, *wavelength*)
Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)

- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds (*filename=None*, *view=False*, *extra=0*, *units='nms'*)
Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.
- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.

sparams (*wavelength*)
Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

Double Half Ring

class SiPANN.scee.**DoubleHalfRing** (*width*, *thickness*, *radius*, *gap*, *sw_angle=90*)
This class will create two equally sized halfrings coupling.

It takes in a radius and gap along with usual waveguide parameters. Ports are numbered as:

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)

- **radius** (*float or ndarray*) – Distance from center of ring to middle of waveguide in nm.
- **gap** (*float or ndarray*) – Minimum distance from ring waveguide edge to other ring waveguide edge in nm. (Must be > 100nm)
- **sw_angle** (*float or ndarray, optional*) – Sidewall angle of waveguide from horizontal in degrees (Valid for 80-90 degrees). Defaults to 90.

update (***kwargs*)

Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict (*ports*, *wavelength*)

Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)
- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds (*filename*, *extra=0*, *units='nm'*, *view=False*)

Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.
- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.

sparams (*wavelength*)

Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

Pushed Half Ring

class SiPANN.scee.**AngledHalfRing**(*width*, *thickness*, *radius*, *gap*, *theta*, *sw_angle=90*)
This class will create a halfring resonator with a pushed side.

It takes in a radius and gap along with usual waveguide parameters. Ports are numbered as:

Parameters

- **width** (*float or ndarray*) – Width of the waveguide in nm (Valid for 400nm-600nm)
- **thickness** (*float or ndarray*) – Thickness of waveguide in nm (Valid for 180nm-240nm)
- **radius** (*float or ndarray*) – Distance from center of ring to middle of waveguide in nm.
- **gap** (*float or ndarray*) – Minimum distance from ring waveguide edge to straight waveguide edge in nm. (Must be > 100nm)
- **theta** (*float or ndarray*) – Angle that the straight waveguide is curved in radians (???).
- **sw_angle** (*float or ndarray, optional (Valid for 80-90 degrees)*) –

update(***kwargs*)
Takes in any parameter defined by __init__ and changes it.

Parameters **attribute** (*float or ndarray*) – Included if any device needs to have an attribute changed.

predict(*ports*, *wavelength*)
Predicts the output when light is put in the specified port (see diagram above)

Parameters

- **ports** (*2-tuple*) – Specifies the port coming in and coming out
- **wavelength** (*float or ndarray*) – Wavelength(s) to predict at (Valid for 1450nm-1650nm)
- **extra_arc** (*float, optional*) – Adds phase to compensate for waveguides getting to gap function. Defaults to 0.
- **part** (*{"both", "mag", "ph"}, optional*) – To speed up calculation, can calculate only magnitude (mag), phase (ph), or both. Defaults to both.

Returns **k/t** – The value of the light coming through

Return type complex ndarray

gds(*filename*, *extra=0*, *units='nm'*, *view=False*)
Writes the geometry to the gds file.

Parameters

- **filename** (*str, optional*) – Location to save file to. Defaults to None.
- **extra** (*int, optional*) – Extra straight portion to add to ends of waveguides to make room in simulation (units same as units parameter). Defaults to 0.
- **units** (*{'microns' or 'nms'}, optional*) – Units to save gds file in. Defaults to microns.
- **view** (*bool, optional*) – Whether to visually show gds file. Defaults to False.

sparams (*wavelength*)

Returns scattering parameters.

Runs SCEE to get scattering parameters at wavelength input.

Parameters **wavelength** (*float or ndarray*) – wavelengths to get sparams at (Valid for 1450nm-1650nm)

Returns

- **freq** (*ndarray*) – frequency for s_matrix in Hz, size n (number of wavelength points)
- **s_matrix** (*ndarray*) – size (n,4,4) complex matrix of scattering parameters, in order of passed in wavelengths

4.11 SCEE Integration

These are the wrappers that provide an interface to enable use of all of the SCEE models in simphony photonic toolbox and Lumerical Interconnect. This gives the user multiple options (Interconnect or Simphony) to cascade devices into complex structures.

4.11.1 Interconnect Exporter

SiPANN.scee_int.**export_interconnect**(*sparams*, *wavelength*, *filename*, *clear=True*)

Exports scattering parameters to a file readable by interconnect.

Parameters

- **sparams** (*ndarray*) – Numpy array of size (N, d, d) where N is the number of frequency points and d the number of ports
- **wavelength** (*ndarray*) – Numpy array of wavelengths (in nm, like the rest of SCEE) of size (N)
- **filename** (*string*) – Location to save file
- **clear** (*bool, optional*) – If True, empties the file first. Defaults to True.

4.11.2 Simphony Wrapper

class SiPANN.scee_int.**SimphonyWrapper**(*model*, *sigmas={}*)

Class that wraps SCEE models for use in simphony.

Model passed into class CANNOT have varying geometries, as a device such as this can't be cascaded properly.

Parameters

- **model** (DC) – Chosen compact model from SiPANN.scee module. Can be any model that inherits from the DC abstract class
- **sigmas** (*dict, optional*) – Dictionary mapping parameters to sigma values for use in monte_carlo simulations. Note sigmas should be in values of nm. Defaults to an empty dictionary.

pins = ('n1', 'n2', 'n3', 'n4')
The default pin names of the device

freq_range = (182800279268292.0, 205337300000000.0)
The valid frequency range for this model.

s_parameters(*freq*)
Get the s-parameters of SCEE Model.

Parameters **freq** (*np.ndarray*) – A frequency array to calculate s-parameters over (in Hz).

Returns **s** – Returns the calculated s-parameter matrix.

Return type np.ndarray

monte_carlo_s_parameters(*freq*)
Get the s-parameters of SCEE Model with slightly changed parameters.

Parameters **freq** (*np.ndarray*) – A frequency array to calculate s-parameters over (in Hz).

Returns **s** – Returns the calculated s-parameter matrix.

Return type np.ndarray

regenerate_monte_carlo_parameters()
Varies parameters based on passed in sigma dictionary.

Iterates through sigma dictionary to change each of those parameters, with the mean being the original values found in model.

connect(*component_or_pin: Union[Model, simphony.pins.Pin]*) → simphony.models.Model
Connects the next available (unconnected) pin from this component to the component/pin passed in as the argument.

If a component is passed in, the first available pin from this component is connected to the first available pin from the other component.

disconnect() → None
Disconnects this component from all other components.

static from_file(*filename: str, *, formatter: Optional[simphony.formatters.ModelFormatter] = None*) → simphony.models.Model
Creates a component from a file using the specified formatter.

Parameters

- **filename** – The filename to read from.
- **formatter** – The formatter instance to use.

static from_string(*string: str, *, formatter: Optional[simphony.formatters.ModelFormatter] = None*) → simphony.models.Model
Creates a component from a string using the specified formatter.

Parameters

- **string** – The string to load the component from.
- **formatter** – The formatter instance to use.

interface(*component: simphony.models.Model*) → simphony.models.Model
Interfaces this component to the component passed in by connecting pins with the same names.

Only pins that have been renamed will be connected.

multiconnect(**connections*) → simphony.models.Model
Connects this component to the specified connections by looping through each connection and connecting it with the corresponding pin.

The first connection is connected to the first pin, the second connection to the second pin, etc. If the connection is set to None, that pin is skipped.

See the `connect` method for more information if the connection is a component or a pin.

rename_pins(**names*) → None
Renames the pins for this component.

The first pin is renamed to the first value passed in, the second pin is renamed to the second value, etc.

to_file(*filename: str*, *freqs: numpy.array*, ***, *formatter: Optional[simphony.formatters.ModelFormatter] = None*) → None
Writes this component's scattering parameters to the specified file using the specified formatter.

Parameters

- **filename** – The name of the file to write to.
- **freqs** – The list of frequencies to save data for.
- **formatter** – The formatter instance to use.

to_string(*freqs: numpy.array*, ***, *formatter: Optional[simphony.formatters.ModelFormatter] = None*) → str
Returns this component's scattering parameters as a formatted string.

Parameters

- **freqs** – The list of frequencies to save data for.
- **formatter** – The formatter instance to use.

4.12 SCEE Optimization

These are various functions to perform and aid in using SCEE as an inverse design optimizer.

Do to how fast SCEE is, inverse design of power splitting directional couplers can be achieved via an optimizer. This has been implemented and can be used via the *SiPANN.scee_opt* module, speficially the *make_coupler* function. It implements a global optimization, then a local optimization to best find the ideal coupler.

This is done by defining the length of the coupler and various control points along the coupler as parameters that our optimizer can choose that result in a $kappa$ closest to $kappa_{goal}$. The coupler is then defined using the control points plugged into a Bezier Curve. Note that the Bezier curve defined by the control points is the gap between waveguides, not the geometry of the waveguides themselves. However, since each of these directional couplers is symmetric the inner boundary of the waveguides are just half of the gap.

Further, for our objective function, we compute $kappa$ for a sweep of wavelength points using SCEE, and then calculate the MSE by comparing it to $kappa_{goal}$. Various constraints are also put on the coupler, like ensuring the edges of the coupler are far enough apart and making sure there's no bends that are too sharp. To learn more about the process, see [INSERT PAPER WHEN PUBLISHED].

4.12.1 Optimizer and Utilities

Inverse Design Optimizer

SiPANN.scee_opt.**make_coupler**(*goalK=0.4*, *arrayK=None*, *waveSweep=array([1500., 1533.33333333, 1566.66666667, 1600.])*, *gapN=16*, *algo=35*, *edgeN=8*, *plot=False*, *collectData=False*, *width=500*, *thickness=220*, *radius=5000*, *maxiter=None*, *verbose=0*)

Optimizes output from a directional coupler defined by a bezier curve to a specified output magnitude.

Parameters

- **goalK** (*float*) – [0-1] mandatory, unless using arrayK. Desired |kappa|^2 magnitude
- **arrayK** (*ndarray, optional*) – Has to have size (2,). [0-1] can specify a |kappa|^2 magnitude at start and end of wavelength sweep. Defaults to None
- **waveSweep** (*ndarray, optional*) – Sweep of wavelengths to evaluate objective function at. Defaults to `np.linspace(1500,1600,4)`
- **gapN** (*int, optional*) – Number of control points that can vary. Defaults to 16.
- **algo** (*int, optional*) – Optimization algorithm that nlopt uses. Defaults to 35
- **edgeN** (*int, optional*) – Number of control points on each edge that are fixed at gap of 1500 nm. Defaults to 8.
- **plot** (*bool, optional*) – If True then optimization will plot the current coupler at each iteration with the control points. Defaults to False.
- **collectData** (*bool, optional*) – Whether to collect data for couplers of each iteration (could be useful for machine learning and even faster design). Defaults to False.
- **width** (*float, optional*) – Width of waveguides in nm. Defaults to 500.
- **thickness** (*float, optional*) – Thickness of waveguides in nm. Defaults to 220.
- **radius** (*float, optional*) – Radius of allowable curve in directional coupler in nm. Defaults to 5000.
- **maxiter** (*int, optional*) – The number of max iterations to run each of the gloabl and local optimization for. If None, doesn't apply. Defaults to None.
- **verbose** (*int, optional*) – Amount of logging to output. If 0, none. If 1, tqdm bar. If 2, prints all information (can be cumbersome). Defaults to 0.

Returns

- **coupler** (*GapFuncSymmetric*) – The final coupler object from SCEE
- **control_pts** (*ndarray*) – The control points defining bezier curve for gap function (nm)
- **length** (*ndarray*) – The length of the coupler (nm)

Saving Couplers

SiPANN.scee_opt.**save_coupler**(*width*, *thickness*, *control_pts*, *length*, *filename*)

Used to save optimized couplers efficiently.

When used only saves gap points and coupling length into a .npz file. This allows for easy reloading via the functions below.

Parameters

- **width** (*float*) – Width of the waveguide in nm
- **thickness** (*float*) – Thickness of the waveguide in nm
- **control_pts** (*ndarray*) – Gap points defining gap function via bernstein polynomials. Second parameter returned by `make_coupler`.
- **length** (*float*) – Length of corresponding coupler. Third parameter returned by `make_coupler`
- **filename** (*string*) – Name of file to write to.

Loading Couplers

`SiPANN.scee_opt.`**load_coupler**(*filename*)

Used to load optimized couplers efficiently.

Any coupler saved using the `save_coupler` function can be reloaded using this one. It will return an instance of `SiPANN.scee.GapFuncSymmetric`.

Parameters **filename** (*string*) – Location where file is stored.

Returns

- **coupler** (*GapFuncSymmetric*) – Saved coupler
- **length** (*float*) – Length of coupler

Premade Couplers

`SiPANN.scee_opt.`**premade_coupler**(*split*)

Loads premade couplers.

Various splitting ratio couplers have been made and saved. This function reloads them. Note that each of their lengths are different and are also returned for the users info. These have all been designed with waveguide geometry 500nm x 220nm.

Parameters **split** (*int*) – Percent of light coming out cross port. Valid numbers are 10, 20, 30, 40, 50, 100. 100 is a full crossover.

Returns

- **coupler** (*GapFuncSymmetric*) – Designed Coupler
- **length** (*float*) – Length of coupler

4.12.2 Helper Functions

Bernstein Transformation

`SiPANN.scee_opt.`**bernstein_quick**()

Quickly computes the jth bernstein polynomial for the basis of n+1 polynomials.

Parameters

- **n** (*int*) – The number of elements minus one in the basis of berstein polynomials
- **j** (*int*) – The index of bernstein polynomial that needs to be computed
- **t** (*float*) – [0-1] the value at which to compute the polynomial

Returns **test** – Result of computing the jth berstein polynomial at t

Return type float

Bezier Function

SiPANN.scee_opt.**bezier_quick**(*g*, *length*)

Computes the bezier curve for the evenly spaced control points with gaps g.

Parameters

- **g** (*ndarray*) – Numpy array of size (n,) of gap values at each of the control points
- **length** (*float*) – length of the coupler

Returns **result** – {'g': original control points, 'w': length of coupler, 'f': bezier curve function defining gap function, 'df': derivative of gap function, 'd2f': 2nd derivative of gap functions}

Return type dict

Index