Simphony Documentation

Release 0.5.1

Sequoia Ploeg

November 09, 2021

TUTORIALS

1 Introduction to Simphony **3**
- 1.1 Models . . . 3
- 1.2 Instantiating Models . . . 4
- 1.3 Connecting Components . . . 4
- 1.4 Simulation . . . 5

2 Mach-Zehnder Interferometer **7**
- 2.1 Deconstructing the MZI . . . 7
- 2.2 Defining The Circuit . . . 8
- 2.3 Sweep Simulation . . . 9
- 2.4 Monte-Carlo Simulation . . . 10

3 Add-Drop Filters **13**
- 3.1 Deconstruction . . . 13
- 3.2 Factory Design Pattern . . . 14
- 3.3 Defining the Circuit . . . 15
- 3.4 Simulation . . . 17

4 API **19**
- 4.1 simphony . . . 19
- 4.2 simphony.formatters . . . 19
- 4.3 simphony.layout . . . 21
- 4.4 simphony.models . . . 23
- 4.5 simphony.pins . . . 27
- 4.6 simphony.simulation . . . 28
- 4.7 simphony.simulators . . . 39
- 4.8 simphony.tools . . . 46

5 Model Libraries **49**
- 5.1 simphony.libraries.siepic . . . 49

6 Contributing to Simphony **65**
- 6.1 Bug Reporting and Feedback . . . 65
- 6.2 Maintaining and Developing . . . 65
- 6.3 Documentation . . . 65

7 Maintaining and Developing **67**
- 7.1 Setting Up the Environment . . . 67
- 7.2 Testing . . . 68
- 7.3 Make a Pull Request . . . 68
- 7.4 Releasing . . . 69

8 Documenting 71

Python Module Index 73

Index 75

Simphony allows you to define photonic circuits, then run fast simulations on them, all in Python.

- Simphony is free and open-source
- Runs on Windows, MacOS, and Linux
- Uses a SPICE-like method for defining photonic circuits
- Subnetwork growth algorithms, giving 20x speedup over other photonic modeling software
- Includes libraries for circuit components (known as models)
- Provides a simple framework for defining new models

To install Simphony, simply use the following in a Python 3 environment:

```
pip install simphony
```

There are also prebuilt releases available on GitHub.

Note: We recommend installing two libraries, matplotlib and SiPANN, alongside Simphony. Matplotlib provides a way to visualize the results from your simulations, and SiPANN provides additional models for use in your circuits. View the links for installation instructions and find out more.

To get started using Simphony, check out *Introduction to Simphony*. Tutorials and API references are accessible through the sidebar navigation.

CHAPTER

ONE

INTRODUCTION TO SIMPHONY

Before we start this tutorial, you should know the basics of Python. We expect you to have an Python environment set up, with the simphony package installed.

Our goal with this tutorial is to define and simulate a simple circuit. In Simphony, circuits are represented all in a single Python file. We'll go through the typical objects found in every circuit definition, in order.

Note: Simphony uses SPICE concepts–such as components, pins, and nets–to define circuits. This should make Simphony intuitive for all those familiar with SPICE, which is commonly used to define electronic circuits.

1.1 Models

Models are basic, presimulated devices; you will connect a these together as components to build your circuit.

A model has a number of ports for inputs and outputs, known as 'pins' in Simphony, and a range of light frequencies that the model is valid over. Internally, it stores a set of scattering parameters (s-parameters). S-parameters, if you aren't already familiar with them, are matrices used when simulating your circuit. We won't go into depth on them here.

Here's an overview of the Model parent class `simphony.models.Model`.

class simphony.models.Model(*name: str = ''*, ***, *freq_range: Optional[Tuple[Optional[float], Optional[float]]] = None*, *pins: Optional[List[simphony.pins.Pin]] = None*)

The basic element type describing the model for a component with scattering parameters.

Any class that inherits from Model or its subclasses must declare either the pin_count or pins attribute. See Attributes for more info.

freq_range

A tuple of the valid frequency bounds for the element in the order (lower, upper). Defaults to (-infty, infty).

Type ClassVar[Tuple[Optional[float], Optional[float]]]

pin_count

The number of pins for the device. Must be set if pins is not.

Type ClassVar[Optional[int]]

pins

A tuple of all the default pin names of the device. Must be set if pin_count is not.

Type *simphony.pins.PinList*

Note: A basic model has no `__init__()` function. It is only required if the model takes in parameters (width or length, for example) that affect the scattering parameters.

All models in Simphony extend this parent class, but redefine pins, frequencies and s-parameters to match the device they represent.

1.2 Instantiating Models

Before we can use a model in our circuit, we need to instantiate it. When we instantiate a model we call the resulting object a component. The difference between models and components is that we can add any kind of state to a component after it has been instantiated, outside of what the model defines.

Simphony includes a default library of models from the SiEPIC PDK (developed at the University of British Columbia). We might define a couple of models with the following:

```
from simphony.libraries import siepic
component1 = siepic.Waveguide(length=500e-9)
component2 = siepic.Waveguide(length=1500e-9)
```

These are both `Waveguide` components. The model has two pins and a valid frequency range. We pass in parameters when instantiating them, so that `component1` will be a shorter `Waveguide` than `component2`. Thus the two will have differing s-parameters, meaning differing simulation results.

Note: All measurements in Simphony should be in base SI units: instead of nanometer measurements, we will pass in meter measurements when instantiating models (i.e. 500e-9 m instead of 500 nm).

1.3 Connecting Components

The *`simphony.pins.Pin`* class is used as an interface to connect two components in a circuit. As an end user, you should rarely have to interact with pins themselves; instead, there are component methods that will handle connecting pins for you. Let's give an example.

Using our previous two components to demonstrate, the simplest way to connect pins is as follows:

```
component1.connect(component2)
```

This will connect the first unconnected pin on both components. However, if we want the first pin of `component1` to be an input, and instead connect its second pin to `component2` as an output, we have to connect the pins explicitly:

```
component1['pin2'].connect(component2['pin1'])
```

By default, a model instantiates its pins with names 'pin1', 'pin2', etc. Here we specify 'pin2' of `component1` must connect to 'pin1' of `component2`. We can also rename pins for semantic clarity:

```
component1.rename_pins('input', 'output')
component1['output'].connect(component2)
```

Here, we do the same as the previous example, except that we rename the two pins of `component1` to 'input' and 'output', and then connect 'output' to `component2`. We do not need to explicitly specify 'pin1' for `component2`, since that is the first unconnected pin.

With this connection, we now have a rudimentary ‘circuit’ to run simulations on.

1.4 Simulation

simphony.simulators provides a collection of simulators that connect to an input and output pin on a circuit, then perform a subnetwork growth algorithm (a series of matrix operations). The results show us what output light comes out of the circuit for given inputs of light. The simulation process modifies pins and components, so simulators actually copy the circuit they are passed in order to preserve the original circuit.

Let’s run a simple sweep simulation on the circuit we have created:

```
from simphony.simulators import SweepSimulation
simulation = SweepSimulation(1500e-9, 1600e-9)
simulation.multiconnect(component1['input'], component2['pin2'])
result = simulation.simulate()
```

We hooked up our simulator to our circuit, with the ‘input’ pin on `component1` being our input and ‘pin2’ on `component2` being our output. Our sweep simulation passed input light on a range of wavelengths from 1500nm to 1600nm, and now `result` contains what frequencies came out of our circuit. We can use these results however we like.

In order to view the results, we can use the `matplotlib` package to graph our output, but that will be demonstrated in following tutorials. For this tutorial, we’re done!

CHAPTER

TWO

MACH-ZEHNDER INTERFEROMETER

In this tutorial, we'll define and simulate a simple circuit known as a Mach-Zender Interferometer (MZI). This tutorial will walk through the code found in `examples/mzi.py` of the Simphony repository. We expect you to have read the previous tutorial, *Introduction to Simphony*.

2.1 Deconstructing the MZI

In an MZI, light entering the circuit is split and travels down two paths of differing lengths. When the light is recombined, it interferes, and the output is frequency-dependent.

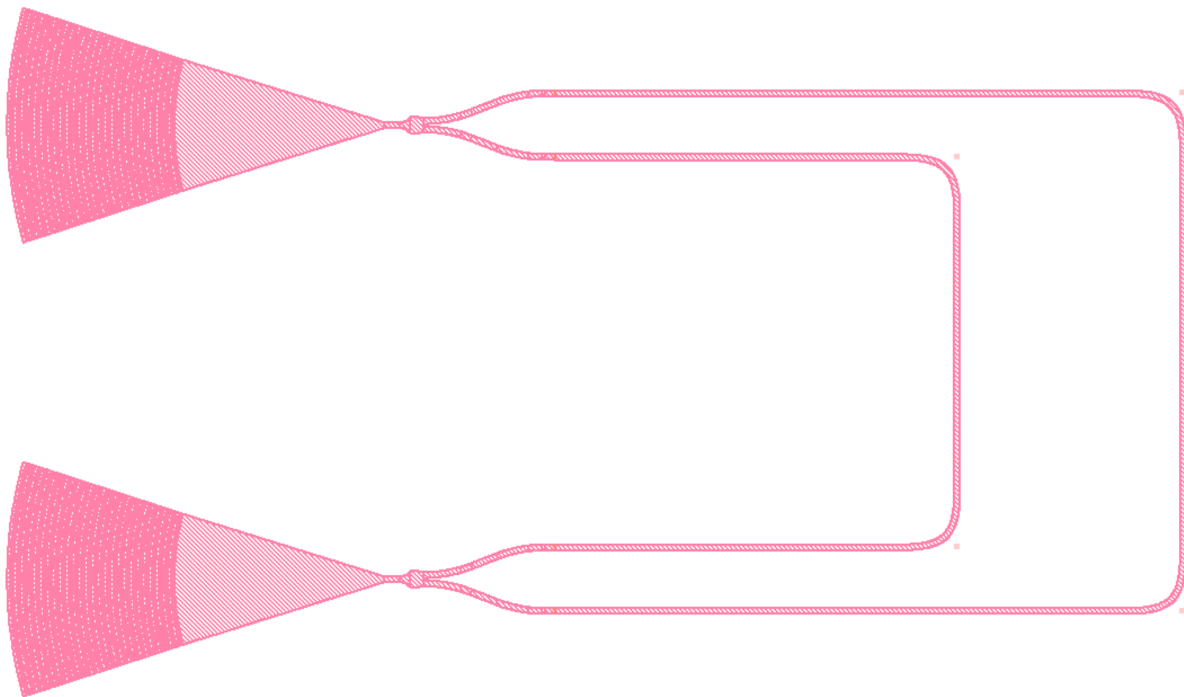
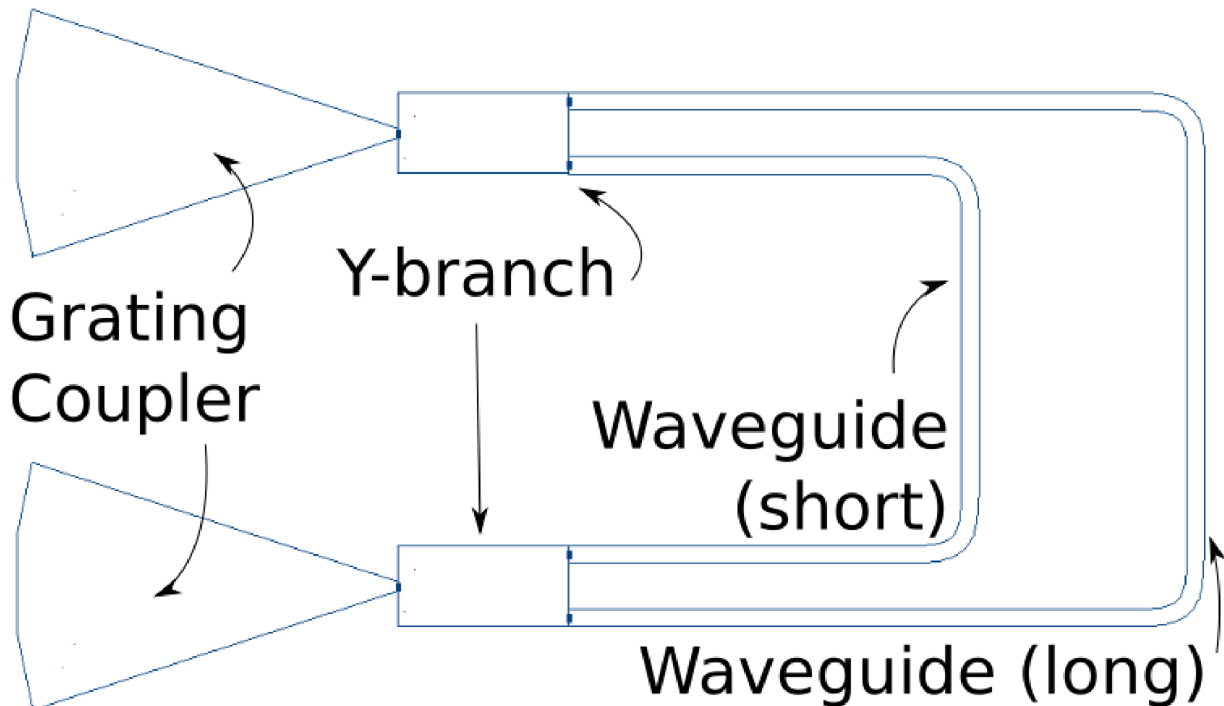


Fig. 1: A basic MZI.

The MZI we'll create can be broken down into constituent parts. Simphony includes models for these constituent parts below:



The grating couplers are the input and output for light in the circuit. The Y-branch can split and recombine light, and because the waveguides which carry light across the circuit are varying length, this produces interference when the light is recombined at the second Y-branch. We can now begin defining our circuit in Simphony using the components we have identified.

2.2 Defining The Circuit

First we need to import the necessary Simphony modules. The `siepic` model library, the `SweepSimulator` and the `MonteCarloSweepSimulator` will be all that we need. We will also import `matplotlib.pyplot`, from outside of Simphony, to view the results of our simulation.

```
import matplotlib.pyplot as plt

from simphony.libraries import siepic
from simphony.simulators import MonteCarloSweepSimulator, SweepSimulator
```

We then create all the components identified earlier. These include the grating couplers, the Y-branches, and the waveguides (which can be defined at any arbitrary length, on the condition that the two lengths are different).

```
gc_input = siepic.GratingCoupler()
y_splitter = siepic.YBranch()
wg_long = siepic.Waveguide(length=150e-6)
wg_short = siepic.Waveguide(length=50e-6)
y_recombiner = siepic.YBranch()
gc_output = siepic.GratingCoupler()
```

Once we have our components, we can connect them. We can reference the diagram above on which connections to make.

```
# next we connect the components to each other
# you can connect pins directly:
y_splitter["pin1"].connect(gc_input["pin1"])

# or connect components with components:
# (when using components to make connections, their first unconnected pin will
# be used to make the connection.)
y_splitter.connect(wg_long)

# or any combination of the two:
y_splitter["pin3"].connect(wg_short)
# y_splitter.connect(wg_short["pin1"])

# when making multiple connections, it is often simpler to use `multiconnect`
# multiconnect accepts components, pins, and None
# if None is passed in, the corresponding pin is skipped
y_recombiner.multiconnect(gc_output, wg_short, wg_long)
```

Note: There are several different ways of connecting components in Simphony, and many are demonstrated in this example, with comments included.

You may also wish to rename pins for better semantics when writing your own circuit. This can be done with `component.rename_pins('new_pin1', 'new_pin2', ...)`, on any of the components we have created.

`rename_pins` can be called on a model itself. This will create default names for any new component you instantiate of that model.

These are all the connections required to define our circuit. The next step will be to run the simulations.

2.3 Sweep Simulation

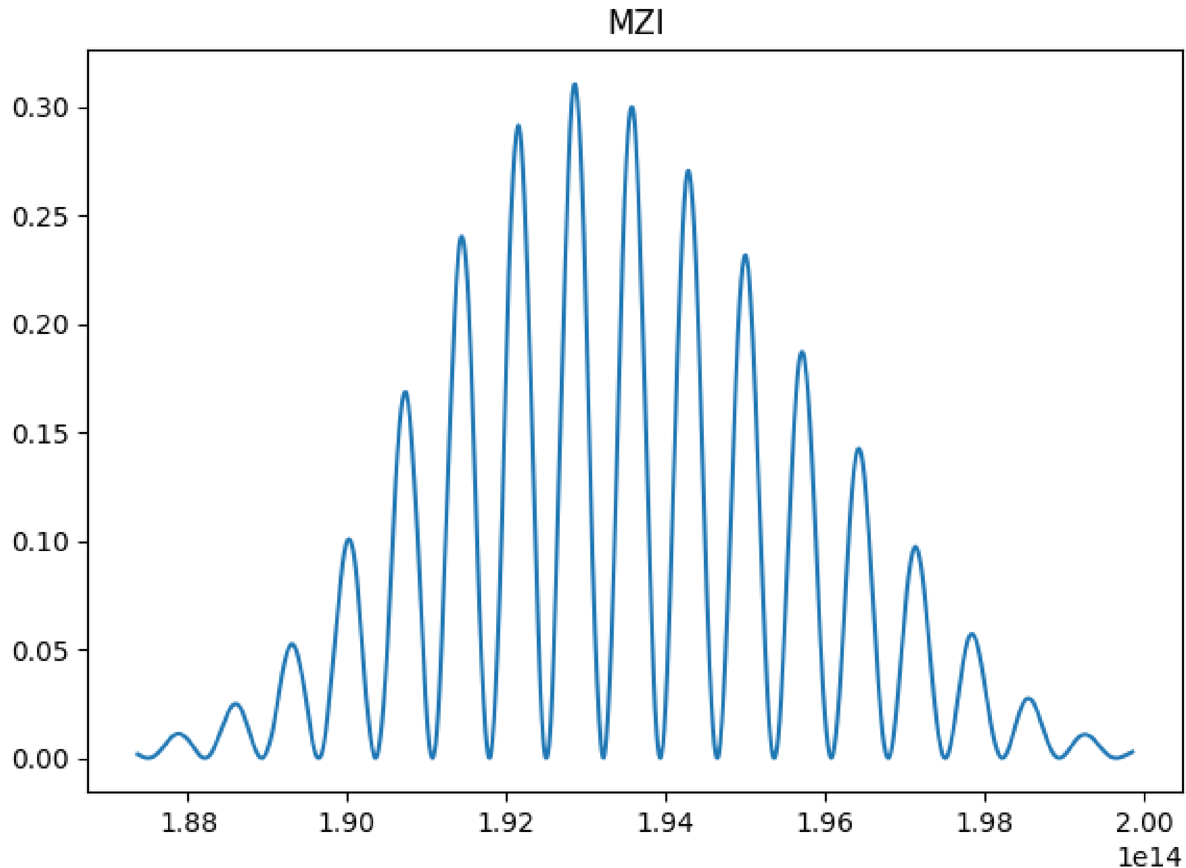
First, we'll do a standard sweep simulation. We define the simulator, the frequency range to run, and then connect it to the input and output of our circuit:

```
simulator = SweepSimulator(1500e-9, 1600e-9)
simulator.multiconnect(gc_input, gc_output)
```

Now we simulate. We hand the results over to `matplotlib` for a graph visualization (see matplotlib docs for how it works).

```
f, p = simulator.simulate()
plt.plot(f, p)
plt.title("MZI")
plt.tight_layout()
plt.show()
```

When you run your MZI python file, it should bring up a graph showing something similar to this:



2.4 Monte-Carlo Simulation

Let's also run a Monte-Carlo simulation. This type of simulation estimates manufacturing variability and margin of error found on real silicon circuits. Disconnect the previous simulator to connect the new simulator:

```
simulator.disconnect()
simulator = MonteCarloSweepSimulator(1500e-9, 1600e-9)
simulator.multiconnect(gc_input, gc_output)
```

Then we run the Monte-Carlo simulation several times, plotting each curve. We will now see several, slightly different curves on our graph due to random variation.

```
results = simulator.simulate(runs=10)
for f, p in results:
  plt.plot(f, p)
```

Finally, we can plot the first simulation again with black, to make sure it displays on top. We want this because the first simulation records results for ideal conditions.

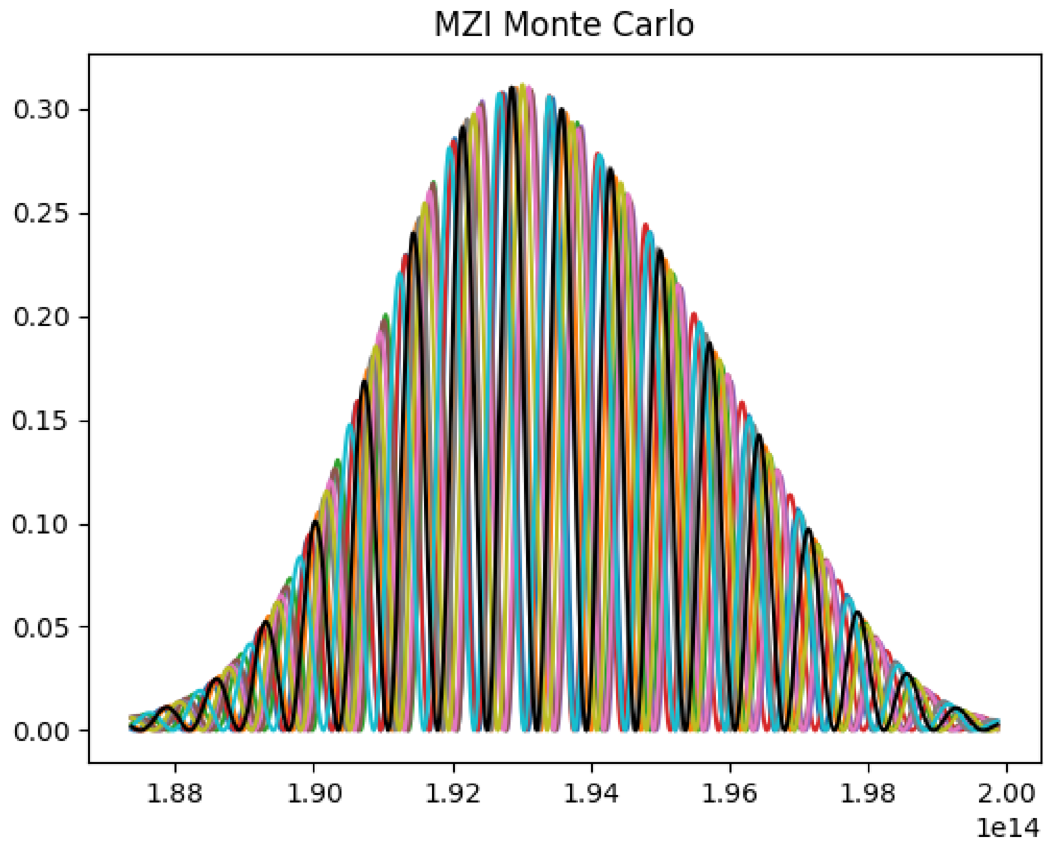
```
f, p = results[0]
plt.plot(f, p, "k")
plt.title("MZI Monte Carlo")
```

(continues on next page)

(continued from previous page)

```
plt.tight_layout()
plt.show()
```

You should see something similar to this graph when you run your MZI now:



We have now defined and simulated our MZI! This completes our tutorial.

CHAPTER THREE

ADD-DROP FILTERS

In this tutorial, we are designing a circuit called an Add-Drop Filter, with a single input and multiple outputs. We'll walk through the code in `examples/filters.py` in the Simphony repo, and we expect you to have already completed the previous tutorial: *Mach-Zehnder Interferometer*.

3.1 Deconstruction

An add-drop filter uses rings of different radii to select specific frequencies from an input waveguide and convey them to an output.

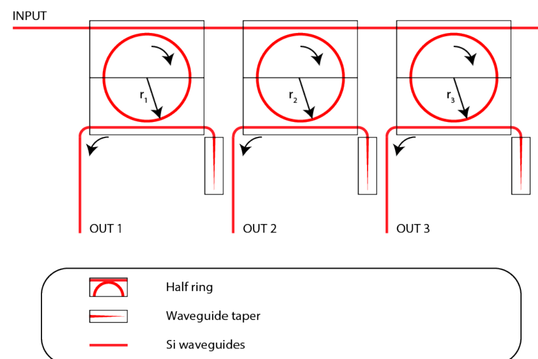


Fig. 1: A sample Add-Drop Filter. The rings all have differing radii.

Light travels through the input waveguide, and some frequencies carry over to the ring waveguides, depending on the radius of the ring. These signals move along the ring until they transfer over to the output waveguides, giving us a reading on what frequencies of light traveled through the input. Light is designed to travel only in one direction after reaching the output waveguides, but we must account for backwards scattering light. We simply add a terminator at the other end of the output waveguides to diffuse any such light.

Notice how the Add-Drop Filter is composed of three similar rings, differing only by their radius:

This single ring resonator can be defined using models from both SiEPIC and SiPANN libraries in Simphony. Instead of defining each model for each ring resonator sequentially, we can use what we call the "factory" design pattern: we will create a method that defines a ring resonator for us.

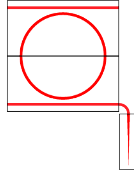


Fig. 2: An isolated, single ring resonator.

3.2 Factory Design Pattern

First, we need to import the libraries we need. The SiEPIC library, the sweep simulator, and matplotlib will be used, just as last tutorial. In addition, we need the SiPANN library. This library of models is not included by default in Simphony, but it integrates well. You will need to install it as shown in the SiPANN docs.

```
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from simphony.libraries import siepic, sipann
from simphony.simulators import SweepSimulator
```

We now create the method that generates a ring resonator for us. We pass in the radius as a parameter, and we are returned a subcircuit, which can be used much the same way a component can.

```
def ring_factory(radius):
  """Creates a full ring (with terminator) from a half ring.

  Resulting pins are ('pass', 'in', 'out').

  Parameters
  ----------
  radius : float
      The radius of the ring resonator, in meters.
  """
  # Have rings for selecting out frequencies from the data line.
  # See SiPANN's model API for argument order and units.
  halfring1 = sipann.HalfRing(500e-9, 220e-9, radius, 100e-9)
  halfring2 = sipann.HalfRing(500e-9, 220e-9, radius, 100e-9)
  terminator = siepic.Terminator()

  halfring1.rename_pins("pass", "midb", "in", "midt")
  halfring2.rename_pins("out", "midt", "term", "midb")

  # the interface method will connect all of the pins with matching names
  # between the two components together
  halfring1.interface(halfring2)
  halfring2["term"].connect(terminator)

  # bundling the circuit as a Subcircuit allows us to interact with it
  # as if it were a component
  return halfring1.circuit.to_subcircuit()
```

Note: In this method, we just demonstrated two new abilities of Simphony that will be of interest to you. First is the `interface` method of a component, another way of connecting components together conveniently. Second is the `to_subcircuit` method. From one of our components, we can get its `circuit`, which includes all components directly or indirectly connected to that first component. We transform that circuit into a Simphony Subcircuit, which behaves similarly to a single component.

Before we construct the full Add-Drop Filter, we can run a simulation on a single ring to make sure everything is behaving as expected.

```
ring1 = ring_factory(10e-6)

simulator = SweepSimulator(1500e-9, 1600e-9)
simulator.multiconnect(ring1["in"], ring1["pass"])

f, t = simulator.simulate(mode="freq")
plt.plot(f, t)
plt.title("10-micron Ring Resonator")
plt.tight_layout()
plt.show()

simulator.disconnect()
```

When you run your python file up to this point, you should see a graph similar to this:

Now that we've created and tested our `ring_factory` method, we can use it to define the Add-Drop Filter.

3.3 Defining the Circuit

Let's create the components we'll use in the circuit:

```
wg_input = siepic.Waveguide(100e-6)
wg_out1 = siepic.Waveguide(100e-6)
wg_connect1 = siepic.Waveguide(100e-6)
wg_out2 = siepic.Waveguide(100e-6)
wg_connect2 = siepic.Waveguide(100e-6)
wg_out3 = siepic.Waveguide(100e-6)
terminator = siepic.Terminator()

ring1 = ring_factory(10e-6)
ring2 = ring_factory(11e-6)
ring3 = ring_factory(12e-6)
```

And then connect each component as seen in the diagram:

```
ring1.multiconnect(wg_connect1, wg_input["pin2"], wg_out1)
ring2.multiconnect(wg_connect2, wg_connect1, wg_out2)
ring3.multiconnect(terminator, wg_connect2, wg_out3)
```

Now we're ready to simulate.

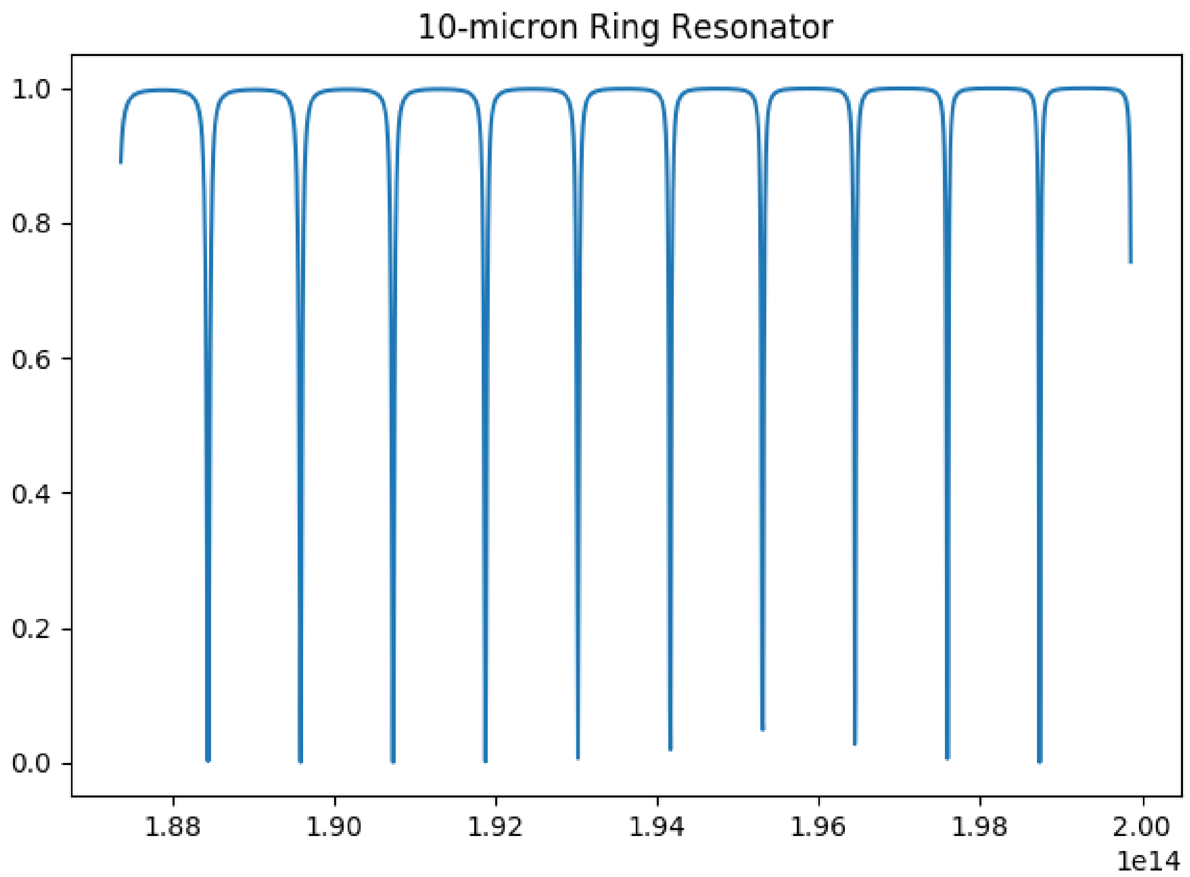


Fig. 3: The through-port frequency response of a 10 micron ring resonator.

3.4 Simulation

We'll run a sweep simulation, but we're reducing the frequency range to 1524.5-1551.15 nm, instead of a full 1500-1600 nm sweep as we have done previously. This will show us a simpler graph of only a few peaks that the filter picks out. We'll be using more advanced matplotlib features here, reference the matplotlib docs on these.

Let's prepare the graph and the simulator to perform simulation:

```
fig = plt.figure(tight_layout=True)
gs = gridspec.GridSpec(1, 3)
ax = fig.add_subplot(gs[0, :2])

simulator = SweepSimulator(1524.5e-9, 1551.15e-9)
simulator.connect(wg_input)
```

Next we simulate each output, and draw a curve for each.

```
# get the results for output 1
simulator.multiconnect(None, wg_out1)
wl, t = simulator.simulate()
ax.plot(wl * 1e9, t, label="Output 1", lw="0.7")

# get the results for output 2
simulator.multiconnect(None, wg_out2)
wl, t = simulator.simulate()
ax.plot(wl * 1e9, t, label="Output 2", lw="0.7")

# get the results for output 3
simulator.multiconnect(None, wg_out3)
wl, t = simulator.simulate()
ax.plot(wl * 1e9, t, label="Output 3", lw="0.7")
```

Then we label our plot.

```
ax.set_ylabel("Fractional Optical Power")
ax.set_xlabel("Wavelength (nm)")
plt.legend(loc="upper right")
```

We could stop here and have a perfectly good plot, but you will notice that one of the peaks will be very small and will be hard to see clearly on this graph. To fix this, we'll add a subplot to our graph to magnify the frequency range of this peak, then simulate and draw each of our outputs on this subplot again.

```
ax = fig.add_subplot(gs[0, 2])

# get the results for output 1
simulator.multiconnect(None, wg_out1)
wl, t = simulator.simulate()
ax.plot(wl * 1e9, t, label="Output 1", lw="0.7")

# get the results for output 2
simulator.multiconnect(None, wg_out2)
wl, t = simulator.simulate()
ax.plot(wl * 1e9, t, label="Output 2", lw="0.7")
```

(continues on next page)

(continued from previous page)

```
# get the results for output 3
simulator.multiconnect(None, wg_out3)
wl, t = simulator.simulate()
ax.plot(wl * 1e9, t, label="Output 3", lw="0.7")

ax.set_xlim(1543, 1545)
ax.set_ylabel("Fractional Optical Power")
ax.set_xlabel("Wavelength (nm)")
fig.align_labels()
```

Finally, we show our plot.

```
plt.show()
```

What you should see when you run your Add-Drop circuit is something like this:

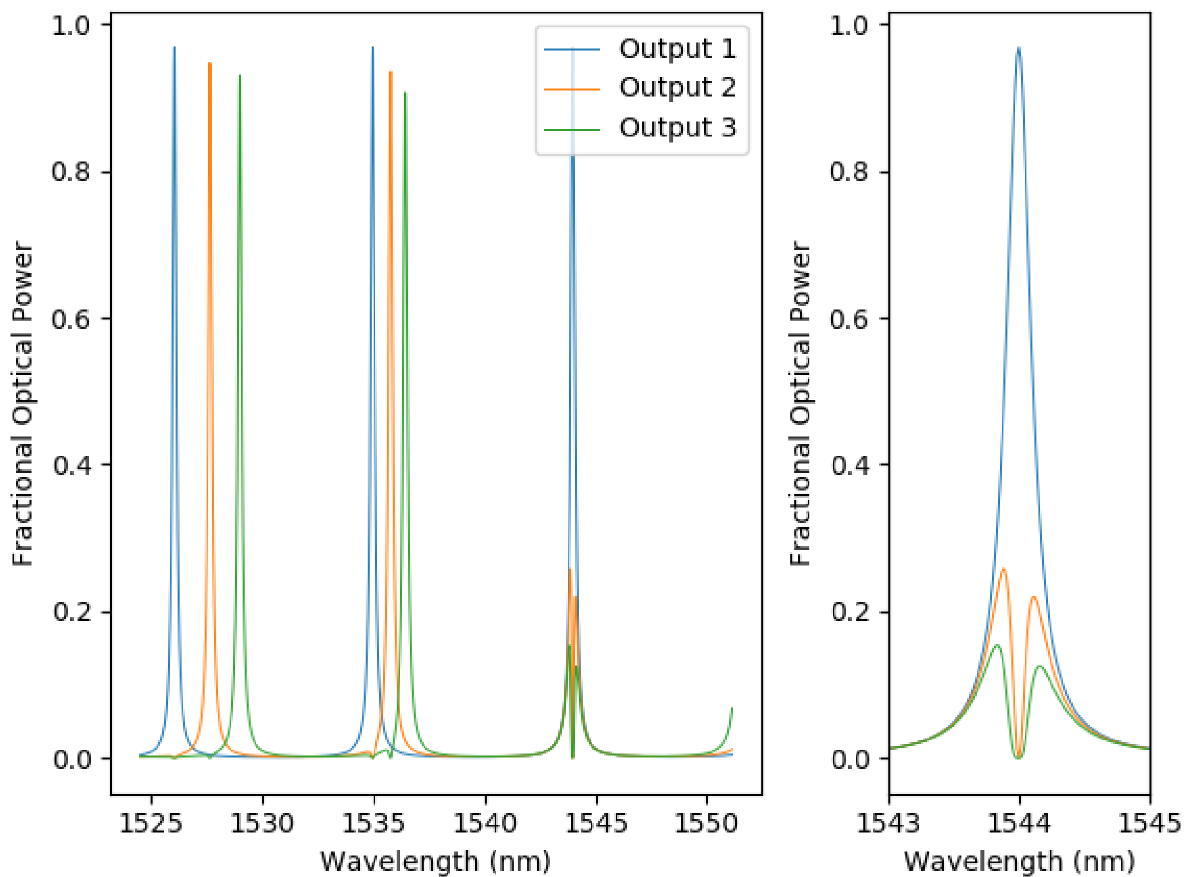


Fig. 4: The response of our designed add-drop filter.

And with that, this tutorial is concluded. For now, this is the last tutorial in the series for learning Simphony. We plan to write more for this series in future, but we hope that this has sufficiently demonstrated the capabilities of Simphony to you. If you wish, you may see the references section to dive into the API for Simphony.

CHAPTER

FOUR

API

4.1 simphony

A Simulator for Photonic circuits

4.2 simphony.formatters

This module contains two types of classes: `ModelFormatters` and `CircuitFormatters`. These classes are used to serialize / unseralize models and circuits.

Specifically, instances of these classes should be used in the `to_file` and `from_file` methods on `Model` and `Circuit`.

class simphony.formatters.**CircuitFormatter**

Bases: `object`

Base circuit formatter class that is extended to provide functionality for converting a circuit to a string and vice-versa.

format(*circuit: Circuit*, *freqs: numpy.array*) → str

Returns a string representation of the circuit.

Parameters **circuit** – The circuit to get a string representation for.

parse(*string: str*) → Circuit

Returns a circuit from the given string.

Parameters **string** – The string to parse.

class simphony.formatters.**CircuitJSONFormatter**

Bases: `object`

This class handles converting a circuit to JSON and vice-versa.

class simphony.formatters.**CircuitSiEPICFormatter**(*pdk=None*)

Bases: `simphony.formatters.CircuitFormatter`

This class saves/loads circuits in the SiEPIC SPICE format.

format(*circuit: Circuit*, *freqs: numpy.array*) → str

Returns a string representation of the circuit.

Parameters **circuit** – The circuit to get a string representation for.

parse(*string: str*) → Circuit

Returns a circuit from the given string.

Parameters `string` – The string to parse.

`class simphony.formatters.`**`JSONDecoder`**

Bases: `json.decoder.JSONDecoder`

JSON Decoder class that handles complex object types.

`decode`(*s*, *_w=<built-in method match of re.Pattern object>*)

Return the Python representation of `s` (a `str` instance containing a JSON document).

`raw_decode`(*s*, *idx=0*)

Decode a JSON document from `s` (a `str` beginning with a JSON document) and return a 2-tuple of the Python representation and the index in `s` where the document ended.

This can be used to decode a JSON document from a string that may have extraneous data at the end.

`class simphony.formatters.`**`JSONEncoder`**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: `json.encoder.JSONEncoder`

JSON Encoder class that handles np.ndarray and complex object types.

`default`(*object*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

`encode`(*o*)

Return a JSON string representation of a Python data structure.

```
>>> from json.encoder import JSONEncoder
>>> JSONEncoder().encode({"foo": ["bar", "baz"]})
'{"foo": ["bar", "baz"]}'
```

`iterencode`(*o*, *_one_shot=False*)

Encode the given object and yield each string representation as available.

For example:

```
for chunk in JSONEncoder().iterencode(bigobject):
    mysocket.write(chunk)
```

`class simphony.formatters.`**`ModelFormatter`**

Bases: `object`

Base model formatter class that is extended to provide functionality for converting a component (model instance) to a string and vice-versa.

format(*component: Model*, *freqs: numpy.array*) → str
Returns a string representation of the component's scattering parameters.

Parameters

- **component** – The component to format.
- **freqs** – The frequencies to get scattering parameters for.

parse(*string: str*) → Model
Returns a component from the given string.

Parameters **string** – The string to parse.

class simphony.formatters.**ModelJSONFormatter**
Bases: *simphony.formatters.ModelFormatter*

The ModelJSONFormatter class formats the model data in a JSON format.

format(*component: Model*, *freqs: numpy.array*) → str
Returns a string representation of the component's scattering parameters.

Parameters

- **component** – The component to format.
- **freqs** – The frequencies to get scattering parameters for.

parse(*string: str*) → Model
Returns a component from the given string.

Parameters **string** – The string to parse.

4.3 simphony.layout

This module contains the `Circuit` object. The `Circuit` object acts as a sorted set that contains components. As components connect/disconnect to each other, they will make sure that they belong to the same `Circuit` instance.

class simphony.layout.**Circuit**(*component: Model*)
Bases: `list`

The `Circuit` class keeps an ordered list of components.

The components themselves manage which circuits they belong to as they are connected and disconnected from one another.

append(*object*, /)
Append object to the end of the list.

clear()
Remove all items from list.

copy()
Return a shallow copy of the list.

count(*value*, /)
Return number of occurrences of value.

extend(*iterable*, /)
Extend list by appending elements from the iterable.

static from_file(*filename: str*, *, *formatter: Optional[*simphony.formatters.CircuitFormatter*] = None*) → *simphony.layout.Circuit*

Creates a circuit from a file using the specified formatter.

Parameters

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

get_pin_index(*pin: Pin*) → int

Gets the pin index for the specified pin in the scattering parameters.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

insert(*index*, *object*, */*)

Insert object before index.

property pins: List[Pin]

Returns the pins for the circuit.

pop(*index=-1*, */*)

Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(*value*, */*)

Remove first occurrence of value.

Raises ValueError if the value is not present.

reverse()

Reverse *IN PLACE*.

s_parameters(*freqs: np.array*) → np.ndarray

Returns the scattering parameters for the circuit.

sort(*, *key=None*, *reverse=False*)

Stable sort *IN PLACE*.

to_file(*filename: str*, *freqs: np.array*, *, *formatter: Optional[*simphony.formatters.CircuitFormatter*] = None*) → None

Writes a string representation of this circuit to a file.

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_subcircuit(*name: str = ''*, ***kwargs*) → Subcircuit

Converts this circuit into a subcircuit component for easy re-use in another circuit.

4.4 simphony.models

This module contains the `Model` and `Subcircuit` classes. The `Model` class is the base class for all models. The `Subcircuit` class is where the subnetwork growth algorithm takes place.

Instances of models are components. As components are connected to each other, they form a circuit. There are three ways to connect components:

1. `comp1_or_pin.connect(comp2_or_pin)`
2. `comp1.multiconnect(comp_or_pin, comp_or_pin, ...)`
3. `comp1.interface(comp2)`

class simphony.models.**Model**(*name: str = ''*, ***, *freq_range: Optional[Tuple[Optional[float], Optional[float]]] = None*, *pins: Optional[List[simphony.pins.Pin]] = None*)

Bases: `object`

The basic element type describing the model for a component with scattering parameters.

Any class that inherits from Model or its subclasses must declare either the pin_count or pins attribute. See Attributes for more info.

freq_range

A tuple of the valid frequency bounds for the element in the order (lower, upper). Defaults to (-infty, infty).

Type ClassVar[Tuple[Optional[float], Optional[float]]]

pin_count

The number of pins for the device. Must be set if pins is not.

Type ClassVar[Optional[int]]

pins

A tuple of all the default pin names of the device. Must be set if pin_count is not.

Type *simphony.pins.PinList*

connect(*component_or_pin: Union[simphony.models.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.

monte_carlo_s_parameters(*freqs: numpy.array*) → numpy.ndarray

Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters **freqs** (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

multiconnect(**connections: Optional[Union[*simphony.models.Model, simphony.pins.Pin*]]*) → *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.

regenerate_monte_carlo_parameters() → None

Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested

for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises `NotImplementedError` – Raised if the subclassing element doesn't implement this function.

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.

class simphony.models.**Subcircuit**(*circuit:* simphony.layout.Circuit, *name: str = ''*, *, *permanent: bool = True*, ***kwargs*)
Bases: *simphony.models.Model*

The `Subcircuit` model exposes the `Model` API for a group of connected components.

Any unconnected pins from the underlying components are re-exposed. This requires that unconnected pins have unique names.

classmethod clear_scache() → None
Clears the scattering parameters cache.

connect(*component_or_pin: Union[*simphony.models.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.

monte_carlo_s_parameters(*freqs: numpy.array*) → numpy.ndarray
Returns the Monte Carlo scattering parameters for the subcircuit.

multiconnect(**connections: Optional[Union[*simphony.models.Model, simphony.pins.Pin*]]*) → *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.

regenerate_monte_carlo_parameters() → None
Regenerates parameters used to generate Monte Carlo s-matrices.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray
Returns the scattering parameters for the subcircuit.

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.

4.5 simphony.pins

This module contains the logic for managing pins and their connections. When connections are made, the pins handle letting the components know which in turn makes sure all components belong to the same `Circuit` instance.

class `simphony.pins.Pin`(*component: Model*, *name: str*)

Bases: `object`

This class represents an individual pin on a component.

As pins are connected and disconnected from each other, the components keep track of which circuit they belong to.

connect(*pin_or_component: Union[Pin, Model]*) → None

Connects this pin to the pin/component that is passed in.

If a component instance is passed in, this pin will connect to the first unconnected pin of the component.

disconnect() → None

Disconnects this pin to whatever it is connected to.

rename(*name: str*) → None

Renames the pin.

class `simphony.pins.PinList`(*component_or_pins: Union[list, Model]*, *length: int = 0*)

Bases: `list`

Keeps track and manages the pins in a component.

append(*object*, */*)

Append object to the end of the list.

clear()

Remove all items from list.

copy()

Return a shallow copy of the list.

count(*value*, */*)

Return number of occurrences of value.

extend(*iterable*, */*)

Extend list by appending elements from the iterable.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

insert(*index*, *object*, */*)

Insert object before index.

pop(*index=-1*, */*)

Remove and return item at index (default last).

Raises IndexError if list is empty or index is out of range.

remove(*value*, */*)

Remove first occurrence of value.

Raises ValueError if the value is not present.

rename(**names: str*) → None

Renames the pins for this pinlist.

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

`reverse()`
Reverse *IN PLACE*.

`sort(*, key=None, reverse=False)`
Stable sort *IN PLACE*.

4.6 simphony.simulation

This module contains the simulation context as well as simulation devices to be used within the context. Devices include theoretical sources and detectors.

class `simphony.simulation.Detector(*args, conversion_gain=1, noise=0, **kwargs)`
Bases: `simphony.simulation.SimulationModel`

The base class for all detectors.

When a detector is connected to the circuit, it defines how many outputs are returned from calling the `Simulation.sample` method. This detector only adds one output.

`connect(component_or_pin: Union[simphony.models.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.

`monte_carlo_s_parameters(freqs: numpy.array) → numpy.ndarray`
Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters **freqs** (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

multiconnect(**connections: Optional[Union[simphony.models.Model, simphony.pins.Pin]]*) → *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.

regenerate_monte_carlo_parameters() → None

Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

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.

class simphony.simulation.**DifferentialDetector**(**args*, *monitor_conversion_gain=1*, *monitor_noise=0*, *rf_conversion_gain=1*, *rf_noise=0*, ***kwargs*)

Bases: `simphony.simulation.Detector`

A differential detector takes two connections and provides three outputs to the `Simulation.sample` method.

The outputs are [connection1, connection1 - connection2, connection2]. The first and third outputs are the monitor outputs and the second output is the RF output.

connect(*component_or_pin: Union[simphony.models.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.

monte_carlo_s_parameters(*freqs: numpy.array*) → numpy.ndarray
Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters **freqs** (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

multiconnect(**connections: Optional[Union[simphony.models.Model, simphony.pins.Pin]]*) → *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.

regenerate_monte_carlo_parameters() → None

Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

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.

class simphony.simulation.**Laser**(*args, *coupling_loss=0*, *freq=None*, *phase=0*, *power=0*, *wl=1.55e-06*, **kwargs)
Bases: *simphony.simulation.Source*

A Simphony model for a laser source.

connect(*component_or_pin: Union[simphony.models.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.

freqsweep(*start: float*, *end: float*, *num: int = 500*) → *simphony.simulation.Laser*
Sets the frequencies to sweep during simulation.

Parameters

- **start** – The frequency to start at.
- **end** – The frequency to end at.
- **num** – The number of frequencies to sweep.

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.

monte_carlo_s_parameters(*freqs: numpy.array*) → numpy.ndarray
Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters **freqs** (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

multiconnect(**connections: Optional[Union[simphony.models.Model, simphony.pins.Pin]]*) → *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.

powersweep(*start: float, end: float, num: int = 500*) → *simphony.simulation.Laser*
Sets the powers to sweep during simulation.

Parameters

- **start** – The power to start at.
- **end** – The power to end at.
- **num** – The number of powers to sweep.

regenerate_monte_carlo_parameters() → None
Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray
Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

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.

wlsweep(*start: float*, *end: float*, *num: int = 500*) → *simphony.simulation.Laser*

Sets the wavelengths to sweep during simulation.

Parameters

- **start** – The wavelength to start at.
- **end** – The wavelength to end at.
- **num** – The number of wavelengths to sweep.

class simphony.simulation.**Simulation**(*, *fs: float = 1000000000.0*, *seed: Optional[int] = None*)

Bases: `object`

This class instantiates a simulation context.

Any simulation devices that are instantiated within the context block are managed by the instance of this class.

classmethod get_context() → *simphony.simulation.Simulation*

Gets the current simulation context.

monte_carlo(*flag: bool*) → None

Sets whether or not to use the Monte Carlo scattering parameters.

Parameters **flag** – When True, Monte Carlo scattering parameters will be used. When False, they will not be used.

s_parameters(*freqs: numpy.array*) → numpy.ndarray

Gets the scattering parameters for the specified frequencies.

Parameters **freqs** – The list of frequencies to run simulations for.

sample(*num_samples: int = 1*) → numpy.ndarray

Samples the outputs of the circuit. If more than one sample is requested, noise will be injected into the system. If only one sample is requested, the returned value will be purely theoretical.

Parameters `num_samples` – The number of samples to take. If only one sample is taken, it will be the theoretical value of the circuit. If more than one sample is taken, they will vary based on simulated noise.

`classmethod set_context(`*_context:* simphony.simulation.Simulation`)` → None

Sets the current simulation context.

Parameters `_context` – The current `Simulation` instance.

`class simphony.simulation.SimulationModel(`**args*, ***kwargs*`)`

Bases: *simphony.models.Model*

A Simphony model that is aware of the current Simulation context.

Models that extend this one should automatically connect to the context upon instantiation.

`connect(`*component_or_pin: Union[*simphony.models.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.

`monte_carlo_s_parameters(`*freqs: numpy.array*`)` → numpy.ndarray

Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters `freqs` (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

multiconnect(**connections: Optional[Union[simphony.models.Model, simphony.pins.Pin]]*) → *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.

regenerate_monte_carlo_parameters() → None

Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

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.

class simphony.simulation.**Source**(**args*, ***kwargs*)

Bases: *simphony.simulation.SimulationModel*

A simphony model for a source.

It automatically connects to the current simulation context upon instantiation.

connect(*component_or_pin: Union[*simphony.models.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.

monte_carlo_s_parameters(*freqs: numpy.array*) → numpy.ndarray

Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters **freqs** (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

multiconnect(**connections: Optional[Union[simphony.models.Model, simphony.pins.Pin]]*) → *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.

regenerate_monte_carlo_parameters() → None

Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

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.

4.7 simphony.simulators

This module contains the simulator components. Simulators must be connected to components before simulating. This can be done using the same connection methods that exist on a component.

class simphony.simulators.**MonteCarloSweepSimulator**(*start: float = 1.5e-06*, *stop: float = 1.6e-06*, *num: int = 2000*)

Bases: *simphony.simulators.SweepSimulator*

Wrapper simulator to make it easier to simulate over a range of frequencies while performing Monte Carlo experimentation.

classmethod clear_scache() → None

Clears the scattering parameters cache.

connect(*component_or_pin: Union[*simphony.models.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.

monte_carlo_s_parameters(*freqs: numpy.array*) → numpy.ndarray

Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters **freqs** (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

multiconnect(**connections: Optional[Union[simphony.models.Model, simphony.pins.Pin]]*) → *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.

regenerate_monte_carlo_parameters() → None

Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

simulate(*runs: int = 10, **kwargs*) → Tuple[numpy.array, numpy.array]

Runs the Monte Carlo sweep simulation for the circuit.

Parameters

- **dB** – Returns the power ratios in deciBels when True.
- **mode** – Whether to return frequencies or wavelengths for the corresponding power ratios. Defaults to whatever values were passed in upon instantiation.

- **runs** – The number of Monte Carlo iterations to run (default 10).

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.

class simphony.simulators.**Simulator**(*name: str = ''*, **, *freq_range: Optional[Tuple[Optional[float], Optional[float]]] = None*, *pins: Optional[List[simphony.pins.Pin]] = None*)

Bases: `simphony.models.Model`

Simulator model that can be used to instantiate a simulator.

The first pin of the simulator should be attached to the input of the circuit that you want to simulate. The second pin should be attached to the output.

classmethod **clear_scache**() → None

Clears the scattering parameters cache.

connect(*component_or_pin: Union[simphony.models.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.

monte_carlo_s_parameters(*freqs: numpy.array*) → numpy.ndarray
Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters **freqs** (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

multiconnect(**connections: Optional[Union[*simphony.models.Model, simphony.pins.Pin*]]*) → *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.

regenerate_monte_carlo_parameters() → None
Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray
Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

simulate(*, *dB: bool = False*, *freq: float = 0*, *freqs: Optional[numpy.array] = None*, *s_parameters_method: str = 's_parameters'*) → Tuple[numpy.array, numpy.array]

Simulates the circuit.

Returns the power ratio at each specified frequency.

Parameters

- **dB** – Returns the power ratios in deciBels when True.
- **freq** – The single frequency to run the simulation for. Must be set if freqs is not.
- **freqs** – The list of frequencies to run simulations for. Must be set if freq is not.
- **s_parameters_method** – The method name to call to get the scattering parameters.

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.

class simphony.simulators.**SweepSimulator**(*start: float = 1.5e-06*, *stop: float = 1.6e-06*, *num: int = 2000*)

Bases: *simphony.simulators.Simulator*

Wrapper simulator to make it easier to simulate over a range of frequencies.

classmethod clear_scache() → None

Clears the scattering parameters cache.

connect(*component_or_pin: Union[simphony.models.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.

`monte_carlo_s_parameters`(*freqs: numpy.array*) → numpy.ndarray

Implements the monte carlo routine for the given Model.

If no monte carlo routine is defined, the default behavior returns the result of a call to `s_parameters()`.

Parameters `freqs` (*np.array*) – The frequency range to generate monte carlo s-parameters over.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `monte_carlo_s_parameters` would be (1, 4, 4).

Return type np.ndarray

`multiconnect`(**connections: Optional[Union[*simphony.models.Model, simphony.pins.Pin*]]*) → *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.

`regenerate_monte_carlo_parameters`() → None

Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

rename_pins(**names: str*) → 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.

s_parameters(*freqs: numpy.array*) → numpy.ndarray
Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

simulate(*mode: Optional[str] = None*, ***kwargs*) → Tuple[numpy.array, numpy.array]
Runs the sweep simulation for the circuit.

Parameters

- **dB** – Returns the power ratios in deciBels when True.
- **mode** – Whether to return frequencies or wavelengths for the corresponding power ratios. Defaults to whatever values were passed in upon instantiation. Either 'freq' or 'wl'.

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.

4.8 simphony.tools

This package contains handy functions useful across simphony submodules and to the average user.

simphony.tools.**freq2wl**(*freq*)

Convenience function for converting from frequency to wavelength.

Parameters **freq** (*float*) – The frequency in SI units (Hz).

Returns **wl** – The wavelength in SI units (m).

Return type float

simphony.tools.**interpolate**(*resampled*, *sampled*, *s_parameters*)

Returns the result of a cubic interpolation for a given frequency range.

Parameters

- **output_freq** (*np.ndarray*) – The desired frequency range for a given input to be interpolated to.
- **input_freq** (*np.ndarray*) – A frequency array, indexed matching the given s_parameters.
- **s_parameters** (*np.array*) – S-parameters for each frequency given in input_freq.

Returns **result** – The values of the interpolated function (fitted to the input s-parameters) evaluated at the `output_freq` frequencies.

Return type np.array

simphony.tools.**str2float**(*num*)

Converts a number represented as a string to a float. Can include suffixes (such as 'u' for micro, 'k' for kilo, etc.).

Parameters **num** (*str*) – A string representing a number, optionally with a suffix.

Returns The string converted back to its floating point representation.

Return type float

Raises **ValueError** – If the argument is malformed or the suffix is not recognized.

Examples

```
>>> str2float('14.5c')
0.145
```

Values without suffixes get converted to floats normally.

```
>>> str2float('2.53')
2.53
```

If an unrecognized suffix is present, a `ValueError` is raised.

```
>>> str2float('17.3o')
ValueError: Suffix 'o' in '17.3o' not recognized.
([-+]?[0-9]+[.]?[0-9]*((?:[eE][-+]?[0-9]+)|[a-zA-Z])?)
```

Some floats are represented in exponential notation instead of suffixes, and we can handle those, too:

```
>>> str2float('15.2e-6')
1.52e-7
```

```
>>> str2float('0.4E6')
400000.0
```

`simphony.tools.`**`wl2freq`**(*wl*)

Convenience function for converting from wavelength to frequency.

Parameters **`wl`** (*`float`*) – The wavelength in SI units (m).

Returns **freq** – The frequency in SI units (Hz).

Return type float

CHAPTER

FIVE

MODEL LIBRARIES

5.1 simphony.libraries.siepic

This package contains parameterized models of PIC components from the SiEPIC Electron Beam Lithography Process Development Kit (PDK), which is licensed under the terms of the MIT License.

5.1.1 Terminology

argset

The code and documentation of this module frequently refer to something hereafter known as an "argset". An *argset* is a dictionary of parameters to values.

For example, let's look at the data files available for a y-branch coupler, as available in *source_data/y_branch_source*. The filename has the format:

```
Ybranch_Thickness =220 width=500.sparam
```

When naming argsets, we use lowercase, underescore-delimited words by convention. In this case, our keys are *thickness* and `width and an argset generated from this filename is:

```
{'thickness': '220', 'width': '500'}
```

It is the responsibility of the model processing argsets to convert this to the normalized form used by the model for comparing its parameters to available data files. For example, the y-branch takes the following parameters:

```
class YBranch(SiEPIC_PDK_Base):
    def __init__(self, thickness=220e-9, width=500e-9, polarization='TE'):
        ...
```

Note that *thickness* and *width* are both floats; lengths, in meters. However, the values parsed from the datafile are strings representing lengths in nanometers. The model therefore creates a normalized set of argsets by converting the values to a form that can be compared with the arguments received by *__init__()*.

Suppose we have the following filename:

```
te_ebeam_dc_halfring_straight_gap=30nm_radius=3um_width=520nm_thickness=210nm_
↪CoupleLength=0um.dat
```

An argset generated from this filename would be:

```
{'gap': '30n', 'radius': '3u', 'width': '520n', 'thickness': '210n', 'couple_length': '0u
↪'}
```

A normalized version of the above argset would be:

```
{'gap': 30e-9, 'radius': 3e-6, 'width': 520e-9, 'thickness': 210e-9, 'couple_length': 0.
↪0}
```

normalized

A variation on argset where the values are formatted to be comparable to the attributes stored by the model. This means that while an argset always reads in strings from datafile names, a normalized argset converts the value to whatever value it actually represents (usually floats).

5.1.2 Future Work

Perhaps we should load the .sparam data files only when *s_parameters()* is called, instead of each time when values are changed. If 2+ attributes are changed, that turns into a lot of extra (needless) file loading.

class simphony.libraries.siepic.**BidirectionalCoupler**(*thickness=2.2e-07*, *width=5e-07*, ***kwargs*)

A bidirectional coupler optimized for TE polarized light at 1550 nanometers.

The bidirectional coupler has 4 ports, labeled as pictured. Its efficiently splits light that is input from one port into the two outputs on the opposite side (with a corresponding pi/2 phase shift). Additionally, it efficiently interferes lights from two adjacent inputs, efficiently splitting the interfered signal between the two ports on the opposing side.

Parameters

- **thickness** (*float, optional*) – Waveguide thickness, in meters (default 220 nanometers). Valid values are 210, 220, or 230 nanometers.
- **width** (*float, optional*) – Waveguide width, in meters (default 500 nanometers). Valid values are 480, 500, or 520 nanometers.

on_args_changed()

Callback for when model attributes are changed; updates the stored s-parameters based on current model attributes.

This function is triggered any time an attribute that is in the model's argument list is changed. For example, if the thickness of the model's waveguides are changed, and thickness is a member of the class's *_args_keys*, this function will automatically be called.

This function should operate only on instance attributes and should not accept parameters.

In summary, `on_args_changed()` must do the following things:

1. Disable autoupdate (`suspend_autoupdate()`).
2. Normalize all argsets for comparison with model attributes.
3. Pass the list of normalized argsets to `_get_matched_args()`, which returns a single normalized argset most closely matching the model's attributes.

4. Update the model's attributes with valid values; values for which we actually have simulation data (so, the values in the argset returned by `_get_matched_args()`.
5. Load the s-parameters from file and store them in a way they can later be accessed by *s_parameters()*, a function also implemented on a class-by-class basis.
6. Set the instance attribute `freq_range`; if this is not set, all simulations on circuits incorporating this model will fail.
7. Enable autoupdate (`enable_autoupdate()`).

> **Warning:** This function will silently change parameters to existing values if no matching data file can be found. For example, if some parameter *radius* has a requested value of 17 but there only exists data for values 15 and 20, the value of radius may be forced to 15 without raising any errors.
>
> A change to any attribute results in a call to this function. To avoid an infinite recursive loop, you should call *suspend_autoupdate()* before modifying any instance attributes. At the end of the function, re-enable autoupdate by calling *enable_autoupdate()*.

s_parameters(*freqs*)

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

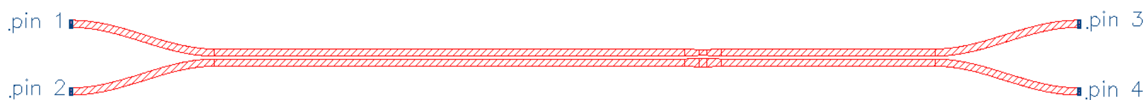
Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

class simphony.libraries.siepic.**DirectionalCoupler**(*gap=2e-07*, *Lc=1e-05*, ***kwargs*)

A directional coupler optimized for TE polarized light at 1550 nanometers.

The directional coupler has 4 ports, labeled as pictured. Its efficiently splits light that is input from one port into the two outputs on the opposite side (with a corresponding pi/2 phase shift). Additionally, it efficiently interferes lights from two adjacent inputs, efficiently splitting the interfered signal between the two ports on the opposing side.



Parameters

- **gap** (*float, optional*) – Coupling gap distance, in meters (default 200 nanometers).
- **Lc** (*float, optional*) – Length of coupler, in meters (default 10 microns).

on_args_changed()

Callback for when model attributes are changed; updates the stored s-parameters based on current model attributes.

This function is triggered any time an attribute that is in the model's argument list is changed. For example, if the thickness of the model's waveguides are changed, and thickness is a member of the class's *_args_keys*,

this function will automatically be called.

This function should operate only on instance attributes and should not accept parameters.

In summary, `on_args_changed()` must do the following things:

1. Disable autoupdate (`suspend_autoupdate()`).
2. Normalize all argsets for comparison with model attributes.
3. Pass the list of normalized argsets to `_get_matched_args()`, which returns a single normalized argset most closely matching the model's attributes.
4. Update the model's attributes with valid values; values for which we actually have simulation data (so, the values in the argset returned by `_get_matched_args()`.
5. Load the s-parameters from file and store them in a way they can later be accessed by *s_parameters()*, a function also implemented on a class-by-class basis.
6. Set the instance attribute `freq_range`; if this is not set, all simulations on circuits incorporating this model will fail.
7. Enable autoupdate (`enable_autoupdate()`).

> **Warning:** This function will silently change parameters to existing values if no matching data file can be found. For example, if some parameter *radius* has a requested value of 17 but there only exists data for values 15 and 20, the value of radius may be forced to 15 without raising any errors.
>
> A change to any attribute results in a call to this function. To avoid an infinite recursive loop, you should call *suspend_autoupdate()* before modifying any instance attributes. At the end of the function, re-enable autoupdate by calling *enable_autoupdate()*.

s_parameters(*freqs*)
Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

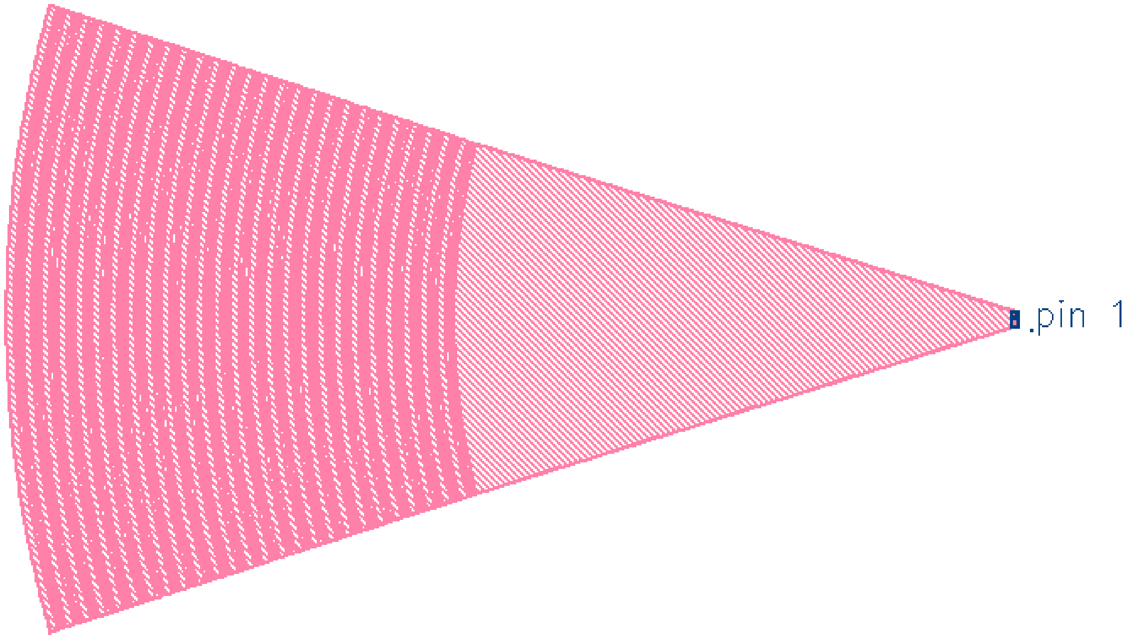
Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

class simphony.libraries.siepic.**GratingCoupler**(*thickness=2.2e-07*, *deltaw=0*, *polarization='TE'*, ***kwargs*)
A grating coupler optimized for TE polarized light at 1550 nanometers.

The grating coupler efficiently couples light from a fiber array positioned above the chip into the circuit. For the TE mode, the angle is -25 degrees [needs citation].



Parameters

- **thickness** (*float, optional*) – The thickness of the grating coupler, in meters (default 220 nanometers). Valid values are 210, 220, or 230 nanometers.
- **deltaw** (*float, optional*) – FIXME: unknown parameter (default 0). Valid values are -20, 0, or 20.
- **polarization** (*str, optional*) – The polarization of light in the circuit. One of 'TE' (default) or 'TM'.

on_args_changed()

Callback for when model attributes are changed; updates the stored s-parameters based on current model attributes.

This function is triggered any time an attribute that is in the model's argument list is changed. For example, if the thickness of the model's waveguides are changed, and thickness is a member of the class's *_args_keys*, this function will automatically be called.

This function should operate only on instance attributes and should not accept parameters.

In summary, `on_args_changed()` must do the following things:

1. Disable autoupdate (`suspend_autoupdate()`).
2. Normalize all argsets for comparison with model attributes.
3. Pass the list of normalized argsets to `_get_matched_args()`, which returns a single normalized argset most closely matching the model's attributes.
4. Update the model's attributes with valid values; values for which we actually have simulation data (so, the values in the argset returned by `_get_matched_args()`.
5. Load the s-parameters from file and store them in a way they can later be accessed by *s_parameters()*, a function also implemented on a class-by-class basis.
6. Set the instance attribute `freq_range`; if this is not set, all simulations on circuits incorporating this model will fail.

7. Enable autoupdate (`enable_autoupdate()`).

> **Warning:** This function will silently change parameters to existing values if no matching data file can be found. For example, if some parameter *radius* has a requested value of 17 but there only exists data for values 15 and 20, the value of radius may be forced to 15 without raising any errors.
>
> A change to any attribute results in a call to this function. To avoid an infinite recursive loop, you should call *suspend_autoupdate()* before modifying any instance attributes. At the end of the function, re-enable autoupdate by calling *enable_autoupdate()*.

s_parameters(*freqs*)

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

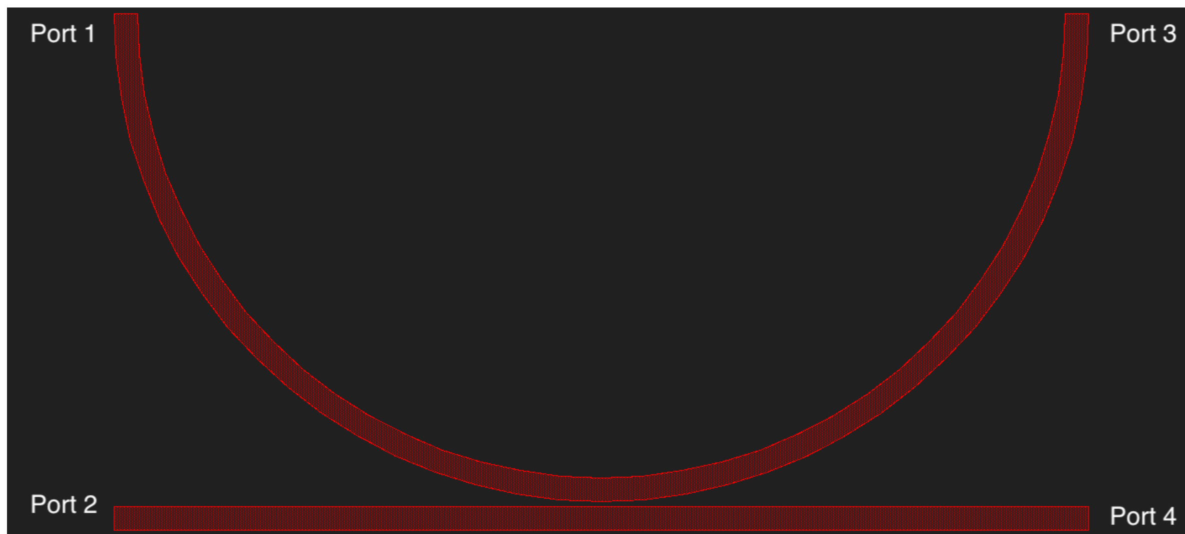
Return type np.ndarray

Raises **NotImplementedError** – Raised if the subclassing element doesn't implement this function.

class simphony.libraries.siepic.**HalfRing**(*gap=3e-08*, *radius=1e-05*, *width=5e-07*, *thickness=2.2e-07*, *couple_length=0.0*, ***kwargs*)

A half-ring resonator optimized for TE polarized light at 1550 nanometers.

The halfring has 4 ports, labeled as pictured.



Parameters

- **gap** (*float, optional*) – Coupling distance between ring and straight waveguide in meters (default 30 nanometers).
- **radius** (*float, optional*) – Ring radius in meters (default 10 microns).
- **width** (*float, optional*) – Waveguide width in meters (default 500 nanometers).

- `thickness` (*float, optional*) – Waveguide thickness in meters (default 220 nanometers).
- `coupler_length` (*float, optional*) – Length of the coupling edge, squares out ring; in meters (default 0).

`on_args_changed()`

Callback for when model attributes are changed; updates the stored s-parameters based on current model attributes.

This function is triggered any time an attribute that is in the model's argument list is changed. For example, if the thickness of the model's waveguides are changed, and thickness is a member of the class's *_args_keys*, this function will automatically be called.

This function should operate only on instance attributes and should not accept parameters.

In summary, `on_args_changed()` must do the following things:

1. Disable autoupdate (`suspend_autoupdate()`).
2. Normalize all argsets for comparison with model attributes.
3. Pass the list of normalized argsets to `_get_matched_args()`, which returns a single normalized argset most closely matching the model's attributes.
4. Update the model's attributes with valid values; values for which we actually have simulation data (so, the values in the argset returned by `_get_matched_args()`.
5. Load the s-parameters from file and store them in a way they can later be accessed by *s_parameters()*, a function also implemented on a class-by-class basis.
6. Set the instance attribute `freq_range`; if this is not set, all simulations on circuits incorporating this model will fail.
7. Enable autoupdate (`enable_autoupdate()`).

> **Warning:** This function will silently change parameters to existing values if no matching data file can be found. For example, if some parameter *radius* has a requested value of 17 but there only exists data for values 15 and 20, the value of radius may be forced to 15 without raising any errors.
>
> A change to any attribute results in a call to this function. To avoid an infinite recursive loop, you should call *suspend_autoupdate()* before modifying any instance attributes. At the end of the function, re-enable autoupdate by calling *enable_autoupdate()*.

`s_parameters`(*freqs*)

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters `freqs` (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises `NotImplementedError` – Raised if the subclassing element doesn't implement this function.

class simphony.libraries.siepic.**SiEPIC_PDK_Base**(***kwargs*)

A base model that includes pre-implemented functions for reading, building, and selecting appropriate .sparam files.

This class is a template to be subclassed and is not supposed to be initialized on its own. Note that the *__init__()* function of subclasses ought to create a normalized set of loaded available parameters that can be compared to the parameters passed in upon construction. Additionally, after finding the most matching parameter set, the parameters should be stored locally for future reference.

By default, a recalculation is triggered to update the s-parameters of the model anytime an instance attribute that is also found in *_args_keys* is modified. To suspend or enable autoupdating, see the functions *enable_autoupdate()* and *suspend_autoupdate()*.

To define a fully working model that subclasses *SiEPIC_PDK_Base*, only *__init__()*, *on_args_changed()*, *s_parameters()*, and the class attributes need to be redefined. A subclass' *__init__()* function should call *super().__init__()* and pass in all parameters that will be saved as attributes. The call to *super* will automatically save them as instance attributes. WARNING: The child class' *__init__()* should NOT save instance attributes itself! They should all be passed to *super*.

Parameters ****kwargs** (*dict*) – The variables that parameterize this component. All are stored as object instance attributes.

args

pins

The default pin names of the device.

Type tuple of str

_base_path

Path to directory containing .sparam files. This should be redefined in every subclass.

Type str

_base_file

A string template that can be filled with argument values to load the appropriate .sparam file. This should be redefined in every subclass.

Type str

_args_keys

The arguments as found in the filename of *.sparam* files. Note that model *kwargs* should match these names, as they are matched by string. The *_base_file* is also filled by matching keywords to the keys defined here. This attribute should be redefined by all subclasses.

Type list of str

_args_trigger_update

Other attributes that, if changed, should also trigger a callback to *on_args_changed()*. Can be an empty list.

Type list of str

_regex

The regular expression that selects available parameters from filenames. This attribute should be redefined by all subclasses.

Type str

Warning: The child class' *__init__()* should NOT save instance attributes itself! They should all be passed to *super*.

property args

A mapping of args (as found in *_args_keys*) to the stored attribute values.

Returns **args** – A property that generates a dictionary of keys to instance values. Keys are specified by *_args_keys*.

Return type dict

enable_autoupdate()

Enables the autoupdate of models when object attributes are modified.

on_args_changed()

Callback for when model attributes are changed; updates the stored s-parameters based on current model attributes.

This function is triggered any time an attribute that is in the model's argument list is changed. For example, if the thickness of the model's waveguides are changed, and thickness is a member of the class's *_args_keys*, this function will automatically be called.

This function should operate only on instance attributes and should not accept parameters.

In summary, `on_args_changed()` must do the following things:

1. Disable autoupdate (`suspend_autoupdate()`).
2. Normalize all argsets for comparison with model attributes.
3. Pass the list of normalized argsets to `_get_matched_args()`, which returns a single normalized argset most closely matching the model's attributes.
4. Update the model's attributes with valid values; values for which we actually have simulation data (so, the values in the argset returned by `_get_matched_args()`.
5. Load the s-parameters from file and store them in a way they can later be accessed by *s_parameters()*, a function also implemented on a class-by-class basis.
6. Set the instance attribute `freq_range`; if this is not set, all simulations on circuits incorporating this model will fail.
7. Enable autoupdate (`enable_autoupdate()`).

Warning: This function will silently change parameters to existing values if no matching data file can be found. For example, if some parameter *radius* has a requested value of 17 but there only exists data for values 15 and 20, the value of radius may be forced to 15 without raising any errors.

A change to any attribute results in a call to this function. To avoid an infinite recursive loop, you should call *suspend_autoupdate()* before modifying any instance attributes. At the end of the function, re-enable autoupdate by calling *enable_autoupdate()*.

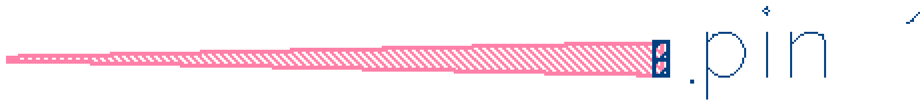
suspend_autoupdate()

Prevents the autoupdate of models when object attributes are modified.

class simphony.libraries.siepic.**Terminator**(*w1=5e-07*, *w2=6e-08*, *L=1e-05*, ***kwargs*)

A terminator component that dissipates light into free space optimized for TE polarized light at 1550 nanometers.

The terminator dissipates excess light into free space. If you have a path where the light doesn't need to be measured but you don't want it reflecting back into the circuit, you can use a terminator to release it from the circuit.



Parameters

- **w1** (*float, optional*) – Width at connecting end in meters (default 500 nanometers).
- **w2** (*float, optional*) – Width at terminating end in meters (default 60 nanometers).
- **L** (*float, optional*) – Length of terminator, in meters (default 10 microns).

`on_args_changed()`

Callback for when model attributes are changed; updates the stored s-parameters based on current model attributes.

This function is triggered any time an attribute that is in the model's argument list is changed. For example, if the thickness of the model's waveguides are changed, and thickness is a member of the class's *_args_keys*, this function will automatically be called.

This function should operate only on instance attributes and should not accept parameters.

In summary, `on_args_changed()` must do the following things:

1. Disable autoupdate (`suspend_autoupdate()`).
2. Normalize all argsets for comparison with model attributes.
3. Pass the list of normalized argsets to `_get_matched_args()`, which returns a single normalized argset most closely matching the model's attributes.
4. Update the model's attributes with valid values; values for which we actually have simulation data (so, the values in the argset returned by `_get_matched_args()`.
5. Load the s-parameters from file and store them in a way they can later be accessed by *s_parameters()*, a function also implemented on a class-by-class basis.
6. Set the instance attribute `freq_range`; if this is not set, all simulations on circuits incorporating this model will fail.
7. Enable autoupdate (`enable_autoupdate()`).

> **Warning:** This function will silently change parameters to existing values if no matching data file can be found. For example, if some parameter *radius* has a requested value of 17 but there only exists data for values 15 and 20, the value of radius may be forced to 15 without raising any errors.
>
> A change to any attribute results in a call to this function. To avoid an infinite recursive loop, you should call *suspend_autoupdate()* before modifying any instance attributes. At the end of the function, re-enable autoupdate by calling *enable_autoupdate()*.

`s_parameters(freqs)`

Returns scattering parameters for the element with its given parameters as declared in the optional `__init__()`.

Parameters **freqs** (*np.array*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range. Its shape should be (the number of frequency points x ports x ports). If the scattering parameters are requested

for only a single frequency, for example, and the device has 4 ports, the shape returned by `s_parameters` would be (1, 4, 4).

Return type np.ndarray

Raises `NotImplementedError` – Raised if the subclassing element doesn't implement this function.

class `simphony.libraries.siepic.`**`Waveguide`**(*length=0.0*, *width=5e-07*, *height=2.2e-07*, *polarization='TE'*, *sigma_ne=0.05*, *sigma_ng=0.05*, *sigma_nd=0.0001*, ***kwargs*)

Model for an waveguide optimized for TE polarized light at 1550 nanometers.

A waveguide easily connects other optical components within a circuit.

Parameters

- **length** (*float*) – Waveguide length in meters (default 0.0 meters).
- **width** (*float, optional*) – Waveguide width in meters (default 500 nanometers).
- **height** (*float, optional*) – Waveguide height in meters (default 220 nanometers).
- **polarization** (*str, optional*) – Polarization of light in the waveguide; one of 'TE' (default) or 'TM'.
- **sigma_ne** (*float, optional*) – Standard deviation of the effective index for monte carlo simulations (default 0.05).
- **sigma_ng** (*float, optional*) – Standard deviation of the group velocity for monte carlo simulations (default 0.05).
- **sigma_nd** (*float, optional*) – Standard deviation of the group dispersion for monte carlo simulations (default 0.0001).

Notes

The *sigma_* values in the parameters are used for monte carlo simulations.

`freq_range: ClassVar[Tuple[Optional[float], Optional[float]]] = (187370000000000.0, 199862000000000.0)`

The valid frequency range for this model.

`monte_carlo_s_parameters`(*freqs*)

Returns a monte carlo (randomized) set of s-parameters.

In this implementation of the monte carlo routine, random values are generated for ne, ng, and nd for each run through of the monte carlo simulation. This means that all waveguide elements throughout a single circuit will have the same (random) ne, ng, and nd values. Hence, there is correlated randomness in the monte carlo parameters but they are consistent within a single circuit.

`on_args_changed`()

Callback for when model attributes are changed; updates the stored s-parameters based on current model attributes.

This function is triggered any time an attribute that is in the model's argument list is changed. For example, if the thickness of the model's waveguides are changed, and thickness is a member of the class's *_args_keys*, this function will automatically be called.

This function should operate only on instance attributes and should not accept parameters.

In summary, `on_args_changed()` must do the following things:

1. Disable autoupdate (`suspend_autoupdate()`).
2. Normalize all argsets for comparison with model attributes.
3. Pass the list of normalized argsets to `_get_matched_args()`, which returns a single normalized argset most closely matching the model's attributes.
4. Update the model's attributes with valid values; values for which we actually have simulation data (so, the values in the argset returned by `_get_matched_args()`.
5. Load the s-parameters from file and store them in a way they can later be accessed by *s_parameters()*, a function also implemented on a class-by-class basis.
6. Set the instance attribute `freq_range`; if this is not set, all simulations on circuits incorporating this model will fail.
7. Enable autoupdate (`enable_autoupdate()`).

> **Warning:** This function will silently change parameters to existing values if no matching data file can be found. For example, if some parameter *radius* has a requested value of 17 but there only exists data for values 15 and 20, the value of radius may be forced to 15 without raising any errors.
>
> A change to any attribute results in a call to this function. To avoid an infinite recursive loop, you should call *suspend_autoupdate()* before modifying any instance attributes. At the end of the function, re-enable autoupdate by calling *enable_autoupdate()*.

`regenerate_monte_carlo_parameters``()`

Regenerates parameters used to generate monte carlo s-matrices.

If a monte carlo method is not implemented for a given model, this method does nothing. However, it can optionally be implemented so that parameters are regenerated once per circuit simulation. This ensures correlation between all components of the same type that reference this model in a circuit. For example, the effective index of a waveguide should not be different for each waveguide in a small circuit; they will be more or less consistent within a single small circuit.

The `MonteCarloSweepSimulation` calls this function once per run over the circuit.

Notes

This function should not accept any parameters, but may act on instance or class attributes.

`s_parameters``(`*freqs*`)`

Get the s-parameters of a waveguide.

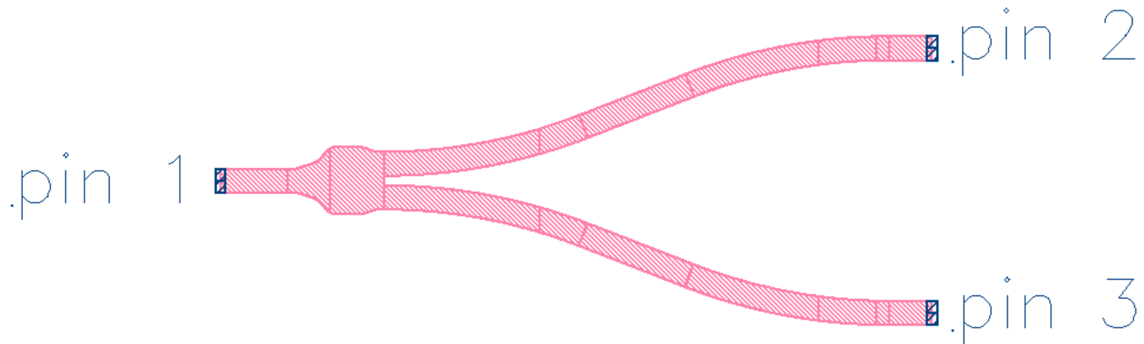
Parameters **`freqs`** (*`float`*) – The array of frequencies to get s parameters for.

Returns **(freqs, s)** – Returns a tuple containing the frequency array, *freqs*, corresponding to the calculated s-parameter matrix, *s*.

Return type tuple

`class simphony.libraries.siepic.`**`YBranch`**`(`*thickness=2.2e-07*, *width=5e-07*, *polarization='TE'*, ***kwargs*`)`

A y-branch efficiently splits the input 50/50 between the two outputs. It can also be used as a combiner if used in the opposite direction, combining and interfering the light from two inputs into the one output.



Parameters

- **thickness** (*float, optional*) – Waveguide thickness, in meters (default 220 nanometers). Valid values are 210, 220, or 230 nanometers.
- **width** (*float, optional*) – Waveguide width, in meters (default 500 nanometers). Valid values are 480, 500, or 520 nanometers.
- **polarization** (*str, optional*) – Polarization of light in the circuit, either 'TE' (default) or 'TM'.

on_args_changed()

Callback for when model attributes are changed; updates the stored s-parameters based on current model attributes.

This function is triggered any time an attribute that is in the model's argument list is changed. For example, if the thickness of the model's waveguides are changed, and thickness is a member of the class's *_args_keys*, this function will automatically be called.

This function should operate only on instance attributes and should not accept parameters.

In summary, `on_args_changed()` must do the following things:

1. Disable autoupdate (`suspend_autoupdate()`).
2. Normalize all argsets for comparison with model attributes.
3. Pass the list of normalized argsets to `_get_matched_args()`, which returns a single normalized argset most closely matching the model's attributes.
4. Update the model's attributes with valid values; values for which we actually have simulation data (so, the values in the argset returned by `_get_matched_args()`.
5. Load the s-parameters from file and store them in a way they can later be accessed by *s_parameters()*, a function also implemented on a class-by-class basis.
6. Set the instance attribute `freq_range`; if this is not set, all simulations on circuits incorporating this model will fail.
7. Enable autoupdate (`enable_autoupdate()`).

Warning: This function will silently change parameters to existing values if no matching data file can be found. For example, if some parameter *radius* has a requested value of 17 but there only exists data for values 15 and 20, the value of radius may be forced to 15 without raising any errors.

A change to any attribute results in a call to this function. To avoid an infinite recursive loop, you should call *suspend_autoupdate()* before modifying any instance attributes. At the end of the function,

re-enable autoupdate by calling *enable_autoupdate()*.

s_parameters(*freqs*)
Returns scattering parameters for the y-branch based on its parameters.

Parameters **freqs** (*np.ndarray*) – The frequency range to get scattering parameters for.

Returns **s** – The scattering parameters corresponding to the frequency range.

Return type np.ndarray

simphony.libraries.siepic.**closest**(*sorted_list*, *value*)
Assumes *sorted_list* is sorted. Returns closest value to *value*.

If two numbers are equally close, return the smallest number.

Parameters

- **sorted_list** (*list of ints or floats*) –
- **value** (*int or float*) –

Returns **closest**

Return type int or float

References

https://stackoverflow.com/a/12141511/11530613

simphony.libraries.siepic.**extract_args**(*strings*, *regex*, *args*)

Parameters

- **strings** (*list of str*) – A list of the strings containing parameters to be extracted.
- **regex** (*str*) – A string representing the regex used to extract the values from the strings.
- **args** (*list of str*) – A list of strings that will be used as keys in the dictionary of values returned. These must be in the same order as the parameters will be extracted by the regex.

Returns **argsets** – A list of all parameter combinations as dictionaries of *args* to values extracted from the string.

Return type list

simphony.libraries.siepic.**get_files_from_dir**(*path*)
Gets the string name of every file in a given directory.

Parameters **path** (*str*) – The absolute path to the directory where the files should be found.

Returns **files** – A list of filenames as strings.

Return type list

simphony.libraries.siepic.**percent_diff**(*ideal*, *actual*)
Calculates the percent error.

Ideally the parameters are of a numeric nature. However, if they are of other types (say, string) a simple value of “1.0” is returned, representing the fact that the two objects are utterly, entirely different.

Parameters

- **ideal** (*float, int, or object*) – The verified accepted value.
- **actual** (*float, int, or object*) – The measured or desired value.

Returns **error** – The percent error of actual from ideal.

Return type float

Notes

Percent error is calculated as (ideal - actual) / ideal

Simphony is primarily developed and maintained by members of the CamachoLab at Brigham Young University. Feedback is welcome: if you find errors or have suggestions for the Simphony project, let us know by raising an issue on Github. If you want to contribute, even better! See *Contributing to Simphony* to learn how.

CHAPTER

SIX

CONTRIBUTING TO SIMPHONY

Simphony is still a growing project, and we appreciate the work of all current and aspiring volunteers! This page will describe some of the ways a volunteer can contribute.

6.1 Bug Reporting and Feedback

Reporting bugs and giving feedback is one of the most accessible ways to help the project, and more of this kind of help is always welcome. If you ever encounter what you believe to be a bug while using Simphony, or if you identify an error in the documentation on this site, look to the Github issues for how to report.

The issues page can also be used to suggest features that you'd like to see added to Simphony. Submit an issue with the correct tag, and plenty of details on how you envision the feature, and we may be able to implement it.

6.2 Maintaining and Developing

If you'd like to work directly in the code of Simphony–and you've identified an area in the codebase that needs updates, improvements, or additions–we'd suggest you create a fork with your changes and submit a pull request on Github. The project contributors will review the pull request as soon as possible, and will work with you to get it merged. For a detailed walkthrough of the development process in Simphony, see *Maintaining and Developing*.

6.3 Documentation

If you've identified problems in the documentation here, and you'd prefer write new documentation or tutorials yourself rather than submit an issue, we've also provided a walkthrough on how to build and develop the docs, separately from the rest of Simphony development. See *Documenting*.

Also, spread the word about Simphony! All kinds of publicity are welcome, from social media to blog posts. And finally, thank you for your excitement about the project! We are glad you've found Simphony as a helpful tool, and for your eagerness to contribute to the project.

CHAPTER

SEVEN

MAINTAINING AND DEVELOPING

This will be a guide in how to contribute to Simphony by writing code. We will go over how to set up a development environment, testing, how to make a pull request, and how to release a version of Simphony. This guide assumes you have a good understanding of Python development and Git.

7.1 Setting Up the Environment

Simphony is a Python 3 package, and so we recommend using the built-in `venv` module for Python 3 to create a virtual environment. From this point forwards, all commands that we provide assume that your new virtual environment is activated.

After creating and activating your virtual environment, go to the *Simphony Github page* and fork the project. You should now have your own copy of the repository on Github. Run:

```
git clone https://github.com/[your username]/simphony.git
```

to download the Github repository onto your machine. You will also want to run:

```
git remote add upstream https://github.com/BYUCamachoLab/simphony.git
```

This will allow you to pull any changes made to the original repository.

Once those steps are complete, run the command:

```
make install
```

in the root directory of Simphony. This will build the development version of Simphony as a package in your virtual environment. The build will update whenever you update Simphony code on your machine, allowing you to use/import the development version in Python files.

You can now start writing your changes to Simphony code. Make sure to always pull changes from upstream before starting development.

7.2 Testing

We expect code to be thoroughly tested before it gets merged with the main Simphony repository.

In Simphony, we include a `tests` directory inside main sections of code, where we write tests using the pytest framework. You should write new tests in these test directories for any new code you develop.

Your code should pass all existing tests before submitting a pull request. Any time you make a commit, a pre-commit hook will run some basic tests and formatters. To run all tests, run the command:

```
make test
```

in the root directory of Simphony. You may also wish to run an individual test, by using:

```
pytest path_to_test/test_file.py
```

Specific formatting issues must also be checked before changes will be merged. You can check all formatting issues using:

```
flake8
```

An additional step you can take with your tests is to measure the test coverage of your code. To do this, install pytest-cov, then run the command:

```
pytest --cov=simphony tests/
```

A final check you may perform before submitting any pull requests is to ensure that your Simphony package will install properly. This is mainly necessary if you intend to release a new version of Simphony. To test this, remove all current installations of Simphony (repeat the command `pip uninstall simphony` until no versions remain) and then run the following commands from Simphony's root directory:

```
python3 setup.py sdist bdist_wheel
pip3 install dist/simphony-[VERSION].tar.gz
```

Note: Doing this will require you to rebuild your development version of Simphony using `make install` again. Before you do this, remove the test version of Simphony you just installed using `pip uninstall simphony` once more.

7.3 Make a Pull Request

After you've made your changes and have done the testing steps above, you're ready to make a pull request. Push your changes to your Github fork, and navigate to the fork's page on Github. You should see a button that will create a pull request for you.

Reviewers will look over your pull request before merging your changes into the main repository, so we expect you to write a clear and concise explanation of your changes attached to your pull request. If your changes are extensive, you will likely need to write more explanation, and perhaps explain the motivation for such extensive changes. Your description of the changes you've made will be used when writing the release notes.

Reviewers will comment on any last minute changes they want to see before merging, such as style and inline documentation, so make sure your code is polished before submitting a pull request. All documentation should follow the numpy doc formatting standard.

Note: If you need to make changes while a pull request is still being reviewed, just push your changes to your fork. The pull request will automatically update to match.

When you submit a pull request, automatic tests will trigger and run through Github's services. These must all pass before your pull request will be accepted. If any fail, click the red cross to pull up a test log, which will help you find out why they failed. Ideally, you will have tested your project before creating the pull request in the first place, so that these tests will not fail.

Finally, when the reviewers believe that the pull request is ready, they will approve the pull request and it will be merged into the main repository.

7.4 Releasing

Most contributors won't have to worry about the release process, since this is up to the core development team. It may still be informative, so we include it here.

The release process is handled by GitHub Actions. When the release script is triggered, it builds the package for Python 3.6-3.8 for Windows, Mac and Linux. It uploads the package to PyPI, creates a GitHub release, and updates the documentation to the most recent stable release.

Before the release, all deprecated code should be removed, and a changelog should be written for the new version. For the changelog, follow the style of previous changelogs when writing. The documentation should be ready for build, see *Documenting* for how to build. Use bump2version to update the version number throughout the project.

Once all of this is complete, run the bash script at `scripts/release`, and the rest will be taken care of automatically.

CHAPTER EIGHT

DOCUMENTING

This is a guide on how to build the documentation web pages (like the ones you are reading right now) from the source available on the Simphony repository. This is useful when writing new documentation, so that you can see how your documentation pages will look.

The docs pages are written in reST. Read the *syntax guide on reST* for more. However, much of the documentation is auto-generated from python docstrings found inline with Simphony code, using the NumPy documentation format.

Since the docs require the Simphony code to build, you will first need to set up a Simphony development environment, as described in the "Setting Up the Environment" section of *Maintaining and Developing*.

Once you have set up your Simphony environment, and you have your virtual environment activated, we will need to install Sphinx. We use Sphinx for generating docs pages: you need to install the latest versions of both the `Sphinx` and `sphinx-rtd-theme` packages with `pip`.

Once Sphinx is installed, you can use the following in the `simphony/docs` directory:

```
make html
```

This will build the documentation pages at `docs/build/html`. Open up any of the HTML pages in that folder to view the documentation on your local machine.

Note: Building the documentation on Windows is not currently supported. (See Sphinx documentation for more information.)

There are also other targets you can build, such as PDF pages instead of HTML pages. Use:

```
make help
```

to see the other targets available.

PYTHON MODULE INDEX

S

simphony, 19
simphony.formatters, 19
simphony.layout, 21
simphony.libraries.siepic, 49
simphony.models, 22
simphony.pins, 26
simphony.simulation, 28
simphony.simulators, 39
simphony.tools, 45

INDEX

Symbols

_args_keys (simphony.libraries.siepic.SiEPIC_PDK_Base attribute), 56
_args_trigger_update (simphony.libraries.siepic.SiEPIC_PDK_Base attribute), 56
_base_file (simphony.libraries.siepic.SiEPIC_PDK_Base attribute), 56
_base_path (simphony.libraries.siepic.SiEPIC_PDK_Base attribute), 56
_regex (simphony.libraries.siepic.SiEPIC_PDK_Base attribute), 56

A

append() (simphony.layout.Circuit method), 21
append() (simphony.pins.PinList method), 27
args (simphony.libraries.siepic.SiEPIC_PDK_Base attribute), 56
args (simphony.libraries.siepic.SiEPIC_PDK_Base property), 56

B

BidirectionalCoupler (class in simphony.libraries.siepic), 50

C

Circuit (class in simphony.layout), 21
CircuitFormatter (class in simphony.formatters), 19
CircuitJSONFormatter (class in simphony.formatters), 19
CircuitSiEPICFormatter (class in simphony.formatters), 19
clear() (simphony.layout.Circuit method), 21
clear() (simphony.pins.PinList method), 27
clear_scache() (simphony.models.Subcircuit class method), 25
clear_scache() (simphony.simulators.MonteCarloSweepSimulator class method), 39
clear_scache() (simphony.simulators.Simulator class method), 41
clear_scache() (simphony.simulators.SweepSimulator class method), 43
closest() (in module simphony.libraries.siepic), 62
connect() (simphony.models.Model method), 23
connect() (simphony.models.Subcircuit method), 25
connect() (simphony.pins.Pin method), 27
connect() (simphony.simulation.Detector method), 28
connect() (simphony.simulation.DifferentialDetector method), 30
connect() (simphony.simulation.Laser method), 32
connect() (simphony.simulation.SimulationModel method), 35
connect() (simphony.simulation.Source method), 37
connect() (simphony.simulators.MonteCarloSweepSimulator method), 39
connect() (simphony.simulators.Simulator method), 41
connect() (simphony.simulators.SweepSimulator method), 43
copy() (simphony.layout.Circuit method), 21
copy() (simphony.pins.PinList method), 27
count() (simphony.layout.Circuit method), 21
count() (simphony.pins.PinList method), 27

D

decode() (simphony.formatters.JSONDecoder method), 20
default() (simphony.formatters.JSONEncoder method), 20
Detector (class in simphony.simulation), 28
DifferentialDetector (class in simphony.simulation), 30
DirectionalCoupler (class in simphony.libraries.siepic), 51
disconnect() (simphony.models.Model method), 23
disconnect() (simphony.models.Subcircuit method), 25
disconnect() (simphony.pins.Pin method), 27
disconnect() (simphony.simulation.Detector method), 28
disconnect() (simphony.simulation.DifferentialDetector method), 30
disconnect() (simphony.simulation.Laser method), 32

disconnect() (simphony.simulation.SimulationModel method), 35
disconnect() (simphony.simulation.Source method), 37
disconnect() (simphony.simulators.MonteCarloSweepSimulator method), 39
disconnect() (simphony.simulators.Simulator method), 41
disconnect() (simphony.simulators.SweepSimulator method), 43

E

enable_autoupdate() (simphony.libraries.siepic.SiEPIC_PDK_Base method), 57
encode() (simphony.formatters.JSONEncoder method), 20
extend() (simphony.layout.Circuit method), 21
extend() (simphony.pins.PinList method), 27
extract_args() (in module simphony.libraries.siepic), 62

F

format() (simphony.formatters.CircuitFormatter method), 19
format() (simphony.formatters.CircuitSiEPICFormatter method), 19
format() (simphony.formatters.ModelFormatter method), 20
format() (simphony.formatters.ModelJSONFormatter method), 21
freq2wl() (in module simphony.tools), 46
freq_range (simphony.libraries.siepic.Waveguide attribute), 59
freq_range (simphony.models.Model attribute), 23
freqsweep() (simphony.simulation.Laser method), 32
from_file() (simphony.layout.Circuit static method), 21
from_file() (simphony.models.Model static method), 23
from_file() (simphony.models.Subcircuit static method), 25
from_file() (simphony.simulation.Detector static method), 28
from_file() (simphony.simulation.DifferentialDetector static method), 30
from_file() (simphony.simulation.Laser static method), 32
from_file() (simphony.simulation.SimulationModel static method), 35
from_file() (simphony.simulation.Source static method), 37
from_file() (simphony.simulators.MonteCarloSweepSimulator static method), 39
from_file() (simphony.simulators.Simulator static method), 41
from_file() (simphony.simulators.SweepSimulator static method), 43
from_string() (simphony.models.Model static method), 23
from_string() (simphony.models.Subcircuit static method), 25
from_string() (simphony.simulation.Detector static method), 28
from_string() (simphony.simulation.DifferentialDetector static method), 30
from_string() (simphony.simulation.Laser static method), 32
from_string() (simphony.simulation.SimulationModel static method), 35
from_string() (simphony.simulation.Source static method), 37
from_string() (simphony.simulators.MonteCarloSweepSimulator static method), 39
from_string() (simphony.simulators.Simulator static method), 41
from_string() (simphony.simulators.SweepSimulator static method), 44

G

get_context() (simphony.simulation.Simulation class method), 34
get_files_from_dir() (in module simphony.libraries.siepic), 62
get_pin_index() (simphony.layout.Circuit method), 22
GratingCoupler (class in simphony.libraries.siepic), 52

H

HalfRing (class in simphony.libraries.siepic), 54

I

index() (simphony.layout.Circuit method), 22
index() (simphony.pins.PinList method), 27
insert() (simphony.layout.Circuit method), 22
insert() (simphony.pins.PinList method), 27
interface() (simphony.models.Model method), 24
interface() (simphony.models.Subcircuit method), 26
interface() (simphony.simulation.Detector method), 28
interface() (simphony.simulation.DifferentialDetector method), 30
interface() (simphony.simulation.Laser method), 32
interface() (simphony.simulation.SimulationModel method), 35
interface() (simphony.simulation.Source method), 37
interface() (simphony.simulators.MonteCarloSweepSimulator method), 39

interface() (*simphony.simulators.Simulator method*), 41
interface() (*simphony.simulators.SweepSimulator method*), 44
interpolate() (*in module simphony.tools*), 46
iterencode() (*simphony.formatters.JSONEncoder method*), 20

J

JSONDecoder (*class in simphony.formatters*), 20
JSONEncoder (*class in simphony.formatters*), 20

L

Laser (*class in simphony.simulation*), 32

M

Model (*class in simphony.models*), 23
ModelFormatter (*class in simphony.formatters*), 20
ModelJSONFormatter (*class in simphony.formatters*), 21
module
    simphony, 19
    simphony.formatters, 19
    simphony.layout, 21
    simphony.libraries.siepic, 49
    simphony.models, 22
    simphony.pins, 26
    simphony.simulation, 28
    simphony.simulators, 39
    simphony.tools, 45
monte_carlo() (*simphony.simulation.Simulation method*), 34
monte_carlo_s_parameters() (*simphony.libraries.siepic.Waveguide method*), 59
monte_carlo_s_parameters() (*simphony.models.Model method*), 24
monte_carlo_s_parameters() (*simphony.models.Subcircuit method*), 26
monte_carlo_s_parameters() (*simphony.simulation.Detector method*), 28
monte_carlo_s_parameters() (*simphony.simulation.DifferentialDetector method*), 30
monte_carlo_s_parameters() (*simphony.simulation.Laser method*), 32
monte_carlo_s_parameters() (*simphony.simulation.SimulationModel method*), 35
monte_carlo_s_parameters() (*simphony.simulation.Source method*), 37
monte_carlo_s_parameters() (*simphony.simulators.MonteCarloSweepSimulator method*), 39
monte_carlo_s_parameters() (*simphony.simulators.Simulator method*), 42
monte_carlo_s_parameters() (*simphony.simulators.SweepSimulator method*), 44
MonteCarloSweepSimulator (*class in simphony.simulators*), 39
multiconnect() (*simphony.models.Model method*), 24
multiconnect() (*simphony.models.Subcircuit method*), 26
multiconnect() (*simphony.simulation.Detector method*), 29
multiconnect() (*simphony.simulation.DifferentialDetector method*), 31
multiconnect() (*simphony.simulation.Laser method*), 33
multiconnect() (*simphony.simulation.SimulationModel method*), 36
multiconnect() (*simphony.simulation.Source method*), 37
multiconnect() (*simphony.simulators.MonteCarloSweepSimulator method*), 40
multiconnect() (*simphony.simulators.Simulator method*), 42
multiconnect() (*simphony.simulators.SweepSimulator method*), 44

O

on_args_changed() (*simphony.libraries.siepic.BidirectionalCoupler method*), 50
on_args_changed() (*simphony.libraries.siepic.DirectionalCoupler method*), 51
on_args_changed() (*simphony.libraries.siepic.GratingCoupler method*), 53
on_args_changed() (*simphony.libraries.siepic.HalfRing method*), 55
on_args_changed() (*simphony.libraries.siepic.SiEPIC_PDK_Base method*), 57
on_args_changed() (*simphony.libraries.siepic.Terminator method*), 58
on_args_changed() (*simphony.libraries.siepic.Waveguide method*), 59
on_args_changed() (*simphony.libraries.siepic.YBranch method*),

61

P

parse() (simphony.formatters.CircuitFormatter method), 19
parse() (simphony.formatters.CircuitSiEPICFormatter method), 19
parse() (simphony.formatters.ModelFormatter method), 21
parse() (simphony.formatters.ModelJSONFormatter method), 21
percent_diff() (in module simphony.libraries.siepic), 62
Pin (class in simphony.pins), 27
pin_count (simphony.models.Model attribute), 23
PinList (class in simphony.pins), 27
pins (simphony.layout.Circuit property), 22
pins (simphony.libraries.siepic.SiEPIC_PDK_Base attribute), 56
pins (simphony.models.Model attribute), 23
pop() (simphony.layout.Circuit method), 22
pop() (simphony.pins.PinList method), 27
powersweep() (simphony.simulation.Laser method), 33

R

raw_decode() (simphony.formatters.JSONDecoder method), 20
regenerate_monte_carlo_parameters() (simphony.libraries.siepic.Waveguide method), 60
regenerate_monte_carlo_parameters() (simphony.models.Model method), 24
regenerate_monte_carlo_parameters() (simphony.models.Subcircuit method), 26
regenerate_monte_carlo_parameters() (simphony.simulation.Detector method), 29
regenerate_monte_carlo_parameters() (simphony.simulation.DifferentialDetector method), 31
regenerate_monte_carlo_parameters() (simphony.simulation.Laser method), 33
regenerate_monte_carlo_parameters() (simphony.simulation.SimulationModel method), 36
regenerate_monte_carlo_parameters() (simphony.simulation.Source method), 38
regenerate_monte_carlo_parameters() (simphony.simulators.MonteCarloSweepSimulator method), 40
regenerate_monte_carlo_parameters() (simphony.simulators.Simulator method), 42
regenerate_monte_carlo_parameters() (simphony.simulators.SweepSimulator method), 44
remove() (simphony.layout.Circuit method), 22
remove() (simphony.pins.PinList method), 27
rename() (simphony.pins.Pin method), 27
rename() (simphony.pins.PinList method), 27
rename_pins() (simphony.models.Model method), 24
rename_pins() (simphony.models.Subcircuit method), 26
rename_pins() (simphony.simulation.Detector method), 29
rename_pins() (simphony.simulation.DifferentialDetector method), 31
rename_pins() (simphony.simulation.Laser method), 33
rename_pins() (simphony.simulation.SimulationModel method), 36
rename_pins() (simphony.simulation.Source method), 38
rename_pins() (simphony.simulators.MonteCarloSweepSimulator method), 40
rename_pins() (simphony.simulators.Simulator method), 42
rename_pins() (simphony.simulators.SweepSimulator method), 45
reverse() (simphony.layout.Circuit method), 22
reverse() (simphony.pins.PinList method), 28

S

s_parameters() (simphony.layout.Circuit method), 22
s_parameters() (simphony.libraries.siepic.BidirectionalCoupler method), 51
s_parameters() (simphony.libraries.siepic.DirectionalCoupler method), 52
s_parameters() (simphony.libraries.siepic.GratingCoupler method), 54
s_parameters() (simphony.libraries.siepic.HalfRing method), 55
s_parameters() (simphony.libraries.siepic.Terminator method), 58
s_parameters() (simphony.libraries.siepic.Waveguide method), 60
s_parameters() (simphony.libraries.siepic.YBranch method), 62
s_parameters() (simphony.models.Model method), 24
s_parameters() (simphony.models.Subcircuit method), 26
s_parameters() (simphony.simulation.Detector method), 29
s_parameters() (simphony.simulation.DifferentialDetector method), 31

s_parameters() (*simphony.simulation.Laser method*), 33
s_parameters() (*simphony.simulation.Simulation method*), 34
s_parameters() (*simphony.simulation.SimulationModel method*), 36
s_parameters() (*simphony.simulation.Source method*), 38
s_parameters() (*simphony.simulators.MonteCarloSweepSimulator method*), 40
s_parameters() (*simphony.simulators.Simulator method*), 42
s_parameters() (*simphony.simulators.SweepSimulator method*), 45
sample() (*simphony.simulation.Simulation method*), 34
set_context() (*simphony.simulation.Simulation class method*), 35
SiEPIC_PDK_Base (*class in simphony.libraries.siepic*), 55
simphony
    module, 19
simphony.formatters
    module, 19
simphony.layout
    module, 21
simphony.libraries.siepic
    module, 49
simphony.models
    module, 22
simphony.pins
    module, 26
simphony.simulation
    module, 28
simphony.simulators
    module, 39
simphony.tools
    module, 45
simulate() (*simphony.simulators.MonteCarloSweepSimulator method*), 40
simulate() (*simphony.simulators.Simulator method*), 43
simulate() (*simphony.simulators.SweepSimulator method*), 45
Simulation (*class in simphony.simulation*), 34
SimulationModel (*class in simphony.simulation*), 35
Simulator (*class in simphony.simulators*), 41
sort() (*simphony.layout.Circuit method*), 22
sort() (*simphony.pins.PinList method*), 28
Source (*class in simphony.simulation*), 37
str2float() (*in module simphony.tools*), 46
Subcircuit (*class in simphony.models*), 25
suspend_autoupdate() (*simphony.libraries.siepic.SiEPIC_PDK_Base method*), 57
SweepSimulator (*class in simphony.simulators*), 43

T

Terminator (*class in simphony.libraries.siepic*), 57
to_file() (*simphony.layout.Circuit method*), 22
to_file() (*simphony.models.Model method*), 25
to_file() (*simphony.models.Subcircuit method*), 26
to_file() (*simphony.simulation.Detector method*), 29
to_file() (*simphony.simulation.DifferentialDetector method*), 31
to_file() (*simphony.simulation.Laser method*), 34
to_file() (*simphony.simulation.SimulationModel method*), 36
to_file() (*simphony.simulation.Source method*), 38
to_file() (*simphony.simulators.MonteCarloSweepSimulator method*), 41
to_file() (*simphony.simulators.Simulator method*), 43
to_file() (*simphony.simulators.SweepSimulator method*), 45
to_string() (*simphony.models.Model method*), 25
to_string() (*simphony.models.Subcircuit method*), 26
to_string() (*simphony.simulation.Detector method*), 30
to_string() (*simphony.simulation.DifferentialDetector method*), 32
to_string() (*simphony.simulation.Laser method*), 34
to_string() (*simphony.simulation.SimulationModel method*), 36
to_string() (*simphony.simulation.Source method*), 38
to_string() (*simphony.simulators.MonteCarloSweepSimulator method*), 41
to_string() (*simphony.simulators.Simulator method*), 43
to_string() (*simphony.simulators.SweepSimulator method*), 45
to_subcircuit() (*simphony.layout.Circuit method*), 22

W

Waveguide (*class in simphony.libraries.siepic*), 59
wl2freq() (*in module simphony.tools*), 47
wlsweep() (*simphony.simulation.Laser method*), 34

Y

YBranch (*class in simphony.libraries.siepic*), 60