Simphony Documentation

Release 0.3.0

Sequoia Ploeg

May 18, 2020
GETTING STARTED

1 Getting Started 3
  1.1 Introduction to Simphony 3
  1.2 Installation 12

2 Using Simphony 13
  2.1 Tutorials 13
  2.2 Simphony Techniques 36
  2.3 Model Libraries 37
  2.4 Simulators 50
  2.5 Simphony Integrations 53

3 Development 55
  3.1 Contributing to Simphony 55

4 Reference 73
  4.1 API 73
  4.2 License 88
  4.3 Glossary 88

Python Module Index 93

Index 95
Simphony, a simulator for photonic circuits, is a fundamental package for designing and simulating photonic integrated circuits with Python.

**Key Features:**

- Free and open-source software provided under the MIT License
- Completely scriptable using Python 3.
- Cross-platform: runs on Windows, MacOS, and Linux.
- Subnetwork growth routines
- A simple, extensible framework for defining photonic component compact models.
- A SPICE-like method for defining photonic circuits.
- Complex simulation capabilities.
- Included model libraries from SiEPIC and SiPANN.
GETTING STARTED

The source repository is hosted on GitHub. Prepackaged wheels of stable versions are in Releases, along with the release history. An additional Changelog is included in the repository.

• Get familiar with Simphony: Introduction to Simphony
• Installation instructions: Installation | Companion Libraries

1.1 Introduction to Simphony

Author  Sequoia Ploeg
Date  May 18, 2020
Copyright  This work is licensed under the MIT License.

This document explains how to use the Simphony Python package for simulation of photonic circuits.

We welcome feedback that would improve the documentation! If you have questions that clarification in the documentation would help resolve, please open an issue at GitHub.

1.1.1 Prerequisites

Before reading this tutorial you should know a bit of Python. If you would like to refresh your memory, take a look at the Python tutorial.

If you wish to work the examples in this tutorial, you must also have some software installed on your computer. Please see Installation for instructions.

1.1.2 Prologue

Simphony aims to be Pythonic, yet pragmatic. Instead of reinventing a new framework for everyone to learn, we build off the concepts that engineers and scientists are already familiar with in electronics: the SPICE way of organizing and defining circuits and connections. In this case, we use much of the same terminology but make it Python friendly (and, in our opinion, improve upon its usability). As a simple example, we use the terminology for components, pins, and nets in a similar manner as an electronics SPICE definition.

Simphony follows Python’s EAFP (easier to ask forgiveness than permission) coding style. This contrasts with the LBYL (look before you leap) style common to other languages. In practice, this means that if, say, a library element component is implemented but is missing attributes, it won’t be noticed until runtime when a call to a nonexistent attribute, perhaps by a simulator, raises an exception.
Python often uses magic methods (also known as “dunder” methods) to implement underlying class functionality. Simphony will sometimes use the same convention, but with what we’ll call “sunder” methods (for single-underscore methods), since Python’s dunder syntax is reserved for future Python features. These methods are typically of no concern to the casual package user; they are used for under-the-hood, behind-the-scenes operations on Simphony objects. Since they represent “private” methods in Python, they won’t appear in the documentation. However, if you begin developing for Simphony, or creating model libraries and plugins, you may need to pay a little more heed to them.

Units throughout Simphony strive to all be in base SI units (unless otherwise specified, see the respective object’s documentation) to avoid ambiguity and confusion. In other words, a length in nanometers will be expressed in meters, etc. This can sometimes lead to not-as-pretty looking values, especially when dealing with sub-wavelength values and frequencies in THz, as is common in silicon photonics. But, it is consistent.

This package initially began as an extension to SiEPIC-Tools, but was broken off into its own independent project as its scope grew and it became large enough to be considered its own stand-alone project. As a result, compatibility with many SiEPIC features has been built into Simphony.

One of the main objectives of the Simphony package is to provide an open-source, concise, independent, and fast simulation tool. By concise, we mean that scripting a simulation does not require many lines of code, imports, or the instantiation of many objects. Independent means that it has very few dependencies and it doesn’t force the user into a specific user interface or proprietary (and hence incompatible) syntax in order to perform simulations. Fast means not recomputing what’s already been simulated, which is accomplished by caching the calculations performed on each component.

1.1.3 Silicon Photonics

Silicon photonics is a rapidly growing industry that uses electronic integrated circuit fabrication technologies to produce industry-grade photonic integrated circuits (PICs) at low cost and high volume. Silicon photonic technologies have been largely driven by the communications industry, but also find applications in sensing, computing, and quantum information processing by enabling high data transmission rates and controlled manipulation of light waves.

As the silicon photonics industry grows and the demand for PICs increases, it is increasingly important for designers to have access to software design tools that can accurately model and simulate PICs in a first-time-right framework. Simulating PICs is a resource- and time-intensive process. Owing to the long wavelengths of photons relative to electrons, photonic device simulation requires solving Maxwell’s equations with far less abstraction than electronic circuit components. Once devices have been simulated and bench-marked, however, compact models representing the phase and amplitude response functions of individual components may be stitched together to form functioning circuits. Although various commercial tools exist to perform these functions, they are often expensive and limited in the variety and type of photonic devices than can be simulated. Furthermore, there is often a lack of standardization among platforms that in many cases prevents interoperability between tools.

Here we present an open-source, software-implemented simulation tool for PICs (documentation and downloads available at github.com/BYUCamachoLab/simphony). Our toolbox, which we name Simphony, provides fast simulations for PICs and allows for the integration of device compact models that may be sourced from a variety of platforms. Simphony also provides the capability to add custom components. This interoperability is achieved by cascading device scattering parameters, or S-parameters, for each component using sub-network growth algorithms, a common practice in microwave/radio-frequency (RF) engineering. Benchmark testing of Simphony against commercial software indicates a speedup of approximately 20x.
1.1.4 Defining Circuits

The fundamental building blocks of a simulation in most SPICE-like programs, and therefore Simphony, are elements, pins, nets, subcircuits, and circuits. A regular SPICE circuit can be fully represented by what is known as a Netlist and stored as a single text file. Simphony circuits are defined simply as Python files, allowing for the persistence of defined circuits and the compatible sharing of designs between computers with a Simphony installation.

Let’s build up our understanding by going through the typical objects that can be found in every circuit definition, in a logical order.

Models

A Model is the basic representation of some designed and presimulated component in Simphony. A simulation library in Simphony is composed of Model definitions, which includes what connections (inputs and outputs) to the model are available, what frequency range the model is valid over, and the scattering parameters of a given model at a given frequency.

Let’s take a look at the documentation for a basic `simphony.elements.Model`.

class simphony.elements.Model

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

Any class that inherits from Model or its subclasses must declare the attributes of an element (see Attributes). Following the general EAFP coding style of Python, errors will only be raised when an unimplemented function is called, not when the class instance is created.

**pins**

A tuple of all the default pin names of the device. Length of default tuple should be equal to the number of ports on the device.

Type tuple of str

**freq_range**

A tuple of the valid frequency bounds for the element in the order (lower, upper). Can be made (-infty, infy) be setting to (None, None).

Type tuple of float

**monte_carlo_s_parameters** *(freq)*

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

- **freq** *(np.ndarray)* – 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

**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.

\textbf{s\_parameters}(\textit{freq})

Returns scattering parameters for the element with its given parameters as declared in the optional \texttt{__init__}().

\begin{itemize}
  \item \textbf{Parameters} \textit{freq} (\texttt{np.ndarray}) – The frequency range to get scattering parameters for.
  \item \textbf{Returns} \textit{s} – The scattering parameters corresponding to the frequency range. Its shape should be \texttt{(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 \texttt{s\_parameters} would be \texttt{(1, 4, 4)}.
  \item \textbf{Return type} \texttt{np.ndarray}
  \item \textbf{Raises} \texttt{NotImplementedError} – Raised if the subclassing element doesn’t implement this function.
\end{itemize}

\textbf{Note:} The base Model class has three default functions but no default \texttt{__init__}() function. It is only required for a model to have an \texttt{__init__}() function if it takes parameters that may affect the s-parameters (e.g. width, length, etc.).

Simphony includes a default model library from the \texttt{SiEPIC EBeam PDK}, developed at the University of British Columbia. Let’s examine one of the models to learn about how they work:

\textbf{class} \texttt{simphony.library.siepic.ebeam\_y\_1550}(\texttt{thickness=2.2e-07, width=5e-07, polarization=TE})

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.

\begin{itemize}
  \item \textbf{thickness} (\texttt{float, optional}) – Waveguide thickness, in meters (default 220 nanometers). Valid values are 210, 220, or 230 nanometers.
  \item \textbf{width} (\texttt{float, optional}) – Waveguide width, in meters (default 500 nanometers). Valid values are 480, 500, or 520 nanometers.
  \item \textbf{polarization} (\texttt{str, optional}) – Polarization of light in the circuit, either ‘TE’ (default) or ‘TM’.
\end{itemize}
The default pin names of the device

`s_parameters(freq)`
Returns scattering parameters for the y-branch based on its parameters.

Parameters `freq` (np.ndarray) – The frequency range to get scattering parameters for.

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

Return type np.ndarray

A simplified version of the code that implements this model is as follows:

class ebeam_y_1550(simphony.elements.Model):
    pins = ('n1', 'n2', 'n3') #: The default pin names of the device

    def __init__(self, thickness=220e-9, width=500e-9, polarization='TE'):
        # Stores values for use in calculating s-parameters later.

    def s_parameters(self, freq):
        # Calculates and returns s-parameters for the given frequency range.

Models define various types of devices. They are not used, however, as instances of those devices in a circuit; instead, instances, or Elements, as they are called in simphony, reference Models to know how they behave.

To use a model in a circuit, we would first instantiate a model with the desired parameters. If we want to use the same model a second time but with different parameters, we’d simply create a second object.

```python
from simphony.library import siepic
y_te = siepic.ebeam_y_1550(thickness=220e-9, width=500e-9, polarization='TE')
y_tm = siepic.ebeam_y_1550(thickness=220e-9, width=500e-9, polarization='TM')
```

**Elements**

An Element represents some discrete component that exists in a layout. It is the physical instantiation of some Model into a circuit. If we consider it from the layout driven design methodology’s perspective, an Element is an object such as a y-branch or a directional coupler that has already been designed and simulated on its own. It has a definite shape and predefined characteristics that allow it to be simply dropped into a layout and operate in a predictable fashion.

Just as a Model is the building block of a simulation, Element objects, or “instances”, are the building blocks of a circuit. Each instance must reference an object of type `simphony.elements.Model` and represents a physical manifestation of some photonic component in the circuit. For example, a waveguide has a single `simphony.elements.Model` which specifies its attributes and S-parameters. However, a circuit may have many waveguides in it, each of differing length. You would therefore instantiate the Waveguide model for all the desired lengths and then associate Elements with the appropriate model. A model can be used for several Elements; if a y-branch with the same parameters is used throughout the circuit, the model only needs to be instantiated once, regardless of how many Elements reference it.

You may never directly instantiate an Element object; when you create instances of Elements in a Subcircuit, the instantiation is handled for you. (Subcircuits will be covered later.) You may, however, provide a unique string name for each element (and this is recommended). If you don’t provide a name, a random one will be generated in the background. If you do provide a name, you’ll be able to retrieve that Element from the circuit later.

Simphony is layout-agnostic; it doesn’t care where components are physically located. However, since everything in Python is an object, other parameters may be stored dynamically within the instance itself, even after its creation. For example, if you were extending Simphony, you may find it useful to include certain key-value pairs, including (but not limited to) layout positioning information. Python, as an interpreted language, contributes to Simphony’s
Simphony Documentation, Release 0.3.0

flexibility when used in conjunction with other programs. For example, Lumerical’s INTERCONNECT, a schematic-driven commercial simulation software, ignores layout and only requires components and connections when simulating a circuit. On the other hand, KLayout with SiEPIC-Tools, created by SiEPIC and UBC, implements a layout-driven approach to designing photonic circuits, exporting locations with the components when generating a netlist. Because Simphony can optionally store any information with its components, including layout information, it can act in either capacity.

Pins (and Pinlists)

Remember the pins attribute from symphony.elements.Model? When elements are created, the pins as defined by the Model are turned into actual symphony.netlist.Pin objects. Pin objects are named according to the names defined by the Model, and are how interactions with pins are handled when defining connections and accessing ports of the circuit once a simulation has been run.

You should never interact with the Pin object itself; rather, only through methods as exposed by Element and Subcircuit objects, typically by using the string name of the Pin. Pins belong to “Pinlists”, which are objects handled internally by Elements. A Pin can only belong to one pinlist at a time, and since the Simulation class manipulates pinlists, any manual intervention can break the definition of a circuit.

You can, however, rename pins for ease of use in making connections. Suppose for example, that you use the same y-branch from earlier to split incoming light between two outputs.

class symphony.library.siepic.ebeam_y_1550(thickness=2.2e-07, width=5e-07, polarization='TE')

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.

![Y-branch diagram](image)

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’.

```python
pins = ('n1', 'n2', 'n3')
```

The default pin names of the device

If we’ve created an Element from the ebeam_y_1550 model, it’ll have pins as follows:
>>> from simphony.netlist import Element
>>> from simphony.library.siepic import ebeam_y_1550
>>> y = ebeam_y_1550()
>>> e = Element(y, name='splitter')

We could rename the pins, once they belong to an element.

One at a time:

```python
>>> e.pins['n1'] = 'input'
>>> e.pins['n2'] = 'output_top'
>>> e.pins['n3'] = 'output_bottom'
```

Or, simultaneously:

```python
>>> e.pins = ('input', 'output_top', 'output_bottom')
```

End result:

```python
>>> e.pins
[<Pin 'input' at <Element 'splitter' at 0x7f9e980a07b8>>,
 <Pin 'output_top' at <Element 'splitter' at 0x7f9e980a07b8>>,
 <Pin 'output_bottom' at <Element 'splitter' at 0x7f9e980a07b8>>]
```

Caution should be taken in naming pins. Multiple pins in a circuit may have the same name, being named uniquely within individual elements. However, consider that when the circuit is fully connected you don’t want any of the remaining external pins to have the same name; it makes identifying outputs unnecessarily difficult. In fact, if you try to access a pin by its string name but multiple pins share the same names, a LookupError is raised complaining that the name is ambiguous as it could match with multiple pins.

**Nets (and Netlists)**

Simphony implements a simple Netlist class. A typical SPICE netlist simply contains a list of components and assigns net numbers to each of its ports, where matching numbers identify connections between components. Simphony alters this slightly, instead maintaining a list of connected pins (which each belong to a specific Element) and dispensing with literal net ID’s.

As a user, you will never interact directly with the netlist, instead defining connections within a subcircuit using syntax explained in a later section.

**Subcircuits**

A Subcircuit is an example of what Simphony will call a compound structure, which the glossary defines as:

**compound structure** Any structure that can be broken down into smaller, simpler parts. A subcircuit is an example of a compound structure; it contains simpler elements (or other compound structures) connected internally to form the overall larger structure.

First, know that a subcircuit can act as a fully-qualified circuit in its own right. In fact, simphony doesn’t even define a Circuit class (for now, at least). Operations are performed on subcircuits themselves.

A circuit can often be broken up into smaller, sometimes reused “subcircuits” that make up the whole design. Subcircuit objects allow us to create these.
Say, for example, that you’d like to create a circuit that cascades several sets of ring resonators of varying radius. Instead of placing lots of halfring elements into a circuit, you can create a standalone subcircuit that represents a ring, and then place the ring subcircuit in your circuit as many times as necessary. This helps reduce complexity and increase abstraction as your circuits grow and the number of elements multiply.

A subcircuit encapsulates a series of element and connection definitions and allows external connections to this circuitry only through the subcircuit’s nodes, also often referred to in Simphony as “externals” (pins not connected within the circuit, thus available for input/output).

Because the internal circuitry is isolated from external circuitry, internal devices and node names with the same names as those external to the subcircuit are neither conflicting nor shorted. In addition, subcircuits can be nested within other subcircuits.

**Putting It Together**

We’ve now discussed separately Models, Elements, Pins, and Subcircuits. Let’s examine how to use these together. We’ll make a rather useless circuit, which simply uses two y-branches to split and input signal and then recombine it. First, we declare the model we want to use for the y-branches. We’ll use the same model for both Element instances.

```python
from simphony.library.siepic import ebeam_y_1550
y_model = ebeam_y_1550()
```

Next, we’ll need a circuit to put the models into. Remember, we don’t have to actually interact with Element objects; when we add models and names to a subcircuit, creating Elements is handled in the background for us.

```python
from simphony.netlist import Subcircuit

circuit = Subcircuit('Example Circuit')
circuit.add({
    (y_model, 'splitter'),
    (y_model, 'recombiner'),
})
```

For ease of manipulation, we’ll rename the ports to something human-readable. The order is determined from the documentation for a y-branch, with the new pin names corresponding to the order of the default pin names.

```python
circuit.elements['splitter'].pins = ('input', 'out1', 'out2')
circuit.elements['recombiner'].pins = ('output', 'in1', 'in2')
```

We can now define the connections for our circuit.

```python
circuit.connect_many({
    ('splitter', 'out1', 'recombiner', 'in1'),
    ('splitter', 'out2', 'recombiner', 'in2'),
})
```

That’s it! Our circuit is fully defined and ready for simulators to come and analyze it.
1.1.5 Simulation

In order to avoid repeating calculations, Simphony caches the scattering parameters for all unique models found in a netlist at simulation time.

The Simulation classes performs the basic sub-network growth matrix operations required to simulate a photonic circuit. Simphony’s simulation module contains various different kinds of simulations for swept simulations, monte carlo simulations, and others. Sub-network growth then cascades all of the instances’ matrices into one matrix.

Simulations take subcircuits as their primary argument. Since the process of cascading the individual s-parameter matrices into a single result matrix is a destructive process and Pins objects are modified in the process, simulators actually create a copy of the circuit given as a parameter so as to not destroy the original circuit, allowing it to be used for other simulations. The only objects that are not fully copied are the instantiated Model classes.

One result of this is that any direct object references (to pins, elements, or subcircuits, for example) are not valid on the resulting subcircuit after a simulation. This is why objects are retrievable by string name; you can use the same string after a simulation to retrieve an object where the actual Python object reference does not point to any object in the simulation results (such as pins).

Simple simulations could be created like so:

```python
from simphony.simulation import SweepSimulation
simulation = SweepSimulation(circuit, 1500e-9, 1600e-9, mode='wl')
result = simulation.simulate()
```

For more details on simphony simulators, see Simulators.

1.1.6 Model Libraries

Simphony includes two basic device libraries. Beyond the default device libraries, it is very easy to define your own collection of component models. Library authors can easily create their own Python modules to be used in simulation by subclassing base components provided by Simphony. This is another way in which Simphony’s capabilities can be easily extended. It also allows simulation results, scripts, and circuits to easily be shared between collaborators or computers, since the entire system is cross-platform, non-proprietary, and the only libraries required are the modules implementing the components used in the circuit and the script defining the circuit.

Read more about how to create custom libraries here.

SIEPIC

The models for the SiEPIC library were created by SiEPIC and the University of British Columbia (UBC) for their Process Design Kit (PDK) used with KLayout. It correlates with the physical component model layouts found in SiEPIC-Tools for KLayout and the S-parameters are the result of FDTD simulations for ideal components.

SiPANN

The second library, which is not preinstalled but can be obtained here, includes components designed by the CamachoLab research group at Brigham Young University (BYU). S-parameters are generated using machine learning techniques. This allows for fast iteration in designing new components as full FDTD simulations don’t need to be run. Additionally, imperfections from manufacturing can be easily simulated using monte carlo techniques.
1.2 Installation

Note: Simphony only supports Python 3.

In most cases, the best way to install Simphony on your system is by installing from PyPI, or “pip”. Updates are regularly pushed as “minor” or “micro” (patch) versions, making upgrading very easy. Installation is as simple as

```
pip install simphony
```

or, depending on your environment setup (for MacOS and Linux),

```
pip3 install simphony
```

If you wish to install outside of pip, you can find prebuilt wheels under GitHub Releases.

1.2.1 Recommended Libraries

**Matplotlib**

The easiest way to visualize the results of simulations is using matplotlib. While not an explicit dependency of Simphony, the tutorials use matplotlib to display what’s happening in a simulation.

1.2.2 Companion Libraries

Several libraries have been developed for use with Simphony and are listed below. If you have developed an open library for use with Simphony, please let us know and we can add it to this list.

**SiPANN**

SiPANN (Silicon Photonics with Artificial Neural Networks) is a library that leverages various machine learning techniques to simulate integrated photonic device circuits, meaning it can very quickly and accurately simulate devices with varying parameters (such as waveguide width or thickness) without having to run a full, slow, FDTD simulation before a designed device can be used in a photonic integrated circuit (PIC) design software such as Simphony.

See the SiPANN documentation for installation instructions. Note that, due to its large number of dependencies, it is a very large package.
Learn the syntax and how to build useful circuits using our simple tutorials.

- **Build simple circuits:** Mach-Zehnder Interferometer | Add-Drop Filters
- **More advanced circuits:** The Green Machine
- **Using Simphony with SiEPIC:** Symphony with SiEPIC
- **Useful design patterns:** The Factory Method Design Pattern
- **Use models from the available libraries:** SiEPIC model library | SiPANN model library

## 2.1 Tutorials

The tutorials are presented as example code for simple and useful circuits with a step-by-step walkthrough explaining how they’re setup. It is hoped that learning by example will be sufficient for gaining an intuition to how Simphony works and how you might construct your own circuits.

### 2.1.1 Creating and Simulating Circuits

**Mach-Zehnder Interferometer**

In this tutorial, we’re going to create a simulation that would predict the frequency response of a Mach-Zehnder Interferometer (MZI).

In an MZI, light entering one port is split and travels down two paths of differing lengths. When the light is recombined, it interferes, producing a frequency-dependent response.

**Code Walkthrough**

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright © Simphony Project Contributors
# Licensed under the terms of the MIT License
# (see simphony/__init__.py for details)
#
# File: mzi.py
```

For this tutorial, we will be using matplotlib and numpy to manipulate and view the results of our simulation.
We’ll need the following modules and objects from simphony:

- **simphony.library.siepic**: The SiEPIC EBeam PDK model library.
- **simphony.netlist.Subcircuit**: We use the Subcircuit object to define our photonic circuits.
- **simphony.simulation.SweepSimulation**: The SweepSimulation takes our circuit and calculates the outputs for a frequency sweep simulation.
- **simphony.simulation.MonteCarloSweepSimulation**: The MonteCarloSweepSimulation will perform several sweep simulations, each time tweaking device parameters to simulate manufacturing variability.

The MZI we’ll create uses only a few simple models. You’ll note that the basic MZI circuit can be broken down into some very basic, repeated “blocks”.

```python
import matplotlib.pyplot as plt
import numpy as np

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

Fig. 1: A basic MZI.
We only need to declare a model for each unique type of block.

1. Grating couplers, for inserting and extracting light from the circuit.

2. Y-branches, for splitting and recombining (and interfering) the light.

3. Waveguides to carry the light from the input to the output. They vary in length in order to produce an interference pattern.
The following snippet declares the grating coupler, the y-branch, and two waveguides; since they have different parameters (length), they’re considered to be unique models.

```python
# Declare the models used in the circuit
gc = siepic.ebeam_gc_te1550()
y = siepic.ebeam_y_1550()
wgl50 = siepic.ebeam_wg_integral_1550(length=150e-6)
wgs50 = siepic.ebeam_wg_integral_1550(length=50e-6)
```

We’ll add all the components into a circuit without worrying about the connections for now. We also give names to the devices as we add them to make the circuit human-readable and for ease in making connections later.

```python
# Create the circuit, add all individual instances
 circuit = Subcircuit('MZI')
 e = circuit.add(
    (gc, 'input'),
    (gc, 'output'),
    (y, 'splitter'),
    (y, 'recombiner'),
    (wgl50, 'wg_long'),
    (wgs50, 'wg_short'),
)
```

Note that `add` returns a list of object references, references to each element added to the subcircuit, where insertion order is preserved.

For ease of making connections, we’ll rename some of the ports. Renaming requires prior knowledge of how ports are laid out on the device. For the SiEPIC library, you can reference the `model library documentation`.

The syntax for renaming pins allows them to be renamed individually:

```python
circuit.elements['input'].pins['n2'] = 'input'
circuit.elements['output'].pins['n2'] = 'output'
```

or simultaneously, the order being the same as the order of the original pin names (reference the model library’s documentation):

```python
circuit.elements['splitter'].pins = ('in1', 'out1', 'out2')
circuit.elements['recombiner'].pins = ('out1', 'in2', 'in1')
```

**Note:** Pins can be renamed for an individual `Element` or for a `Model`. When renamed for an Element, only that Element is affected; if renamed for a Model, all future created Elements that reference that Model will assume the new pin names. See the `Introduction` to learn about the difference between Elements and Models.
Next we define the circuit’s connections, again referencing the pin names (using the default names, unless you’ve renamed the pins). You can use either `connect`, which accepts four parameters:

```
(f first element name, first element port, connecting element name, connecting element _port)
```

or `connect_many`, which takes a list of tuples, each tuple having the same four-argument format.

```
# Circuits can be connected using the elements' string names:
circuit.connect_many(
    ('input', 'n1', 'splitter', 'in1'),
    ('splitter', 'out1', 'wg_long', 'n1'),
    ('splitter', 'out2', 'wg_short', 'n1'),
    ('recombiner', 'in1', 'wg_long', 'n2'),
    ('recombiner', 'in2', 'wg_short', 'n2'),
    ('output', 'n1', 'recombiner', 'out1'),
)
```

Note that connections can be made using array indexing as well. If you know what order your components were inserted into the circuit (insertion order is preserved) or what index the pin you want to connect to is at, you can use those details to form your connections.

```
# or by using the actual object reference.
# circuit.connect(e[0], e[0].pin[0], e[2], e[2].pin[0])
```

At this point, your circuit is defined. This file can serve as a description for a subcircuit that is used in a larger circuit, and can simply be imported using the Python import system (e.g., `from mzi import circuit`).

Running a simulation on our fully-defined circuit is really easy.

```
simulation = SweepSimulation(circuit, 1500e-9, 1600e-9)
result = simulation.simulate()
```

Finally, we can plot the simulation result data. We can access the pins using their string names.

```
f, s = result.data('input', 'output')
plt.plot(f, s)
plt.title("MZI")
plt.tight_layout()
plt.show()
```
We can even run a monte carlo simulation, which simulates manufacturing variability. The data stored at the 0th index, and plotted on top in black, is the ideal values.

```python
# We can run a monte carlo simulation on the circuit, too
simulation = MonteCarloSweepSimulation(circuit, 1500e-9, 1600e-9)
runs = 10
result = simulation.simulate(runs=runs)
for i in range(1, runs + 1):
    f, s = result.data('input', 'output', i, dB=True)
    plt.plot(f, s)

# The data located at the 0 position is the ideal values.
f, s = result.data('input', 'output', 0, dB=True)
plt.plot(f, s, 'k')
plt.title("MZI Monte Carlo")
plt.tight_layout()
plt.show()
```
# Full Code Listing

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright © Simphony Project Contributors
# Licensed under the terms of the MIT License
# (see simphony/__init__.py for details)
#
# File: mzi.py

import matplotlib.pyplot as plt
import numpy as np

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

# Declare the models used in the circuit
gc = siepic.ebeam_gc_te1550()
y = siepic.ebeam_y_1550()
wg150 = siepic.ebeam_wg_integral_1550(length=150e-6)
wg50 = siepic.ebeam_wg_integral_1550(length=50e-6)
```

(continues on next page)
# Create the circuit, add all individual instances

circuit = Subcircuit('MZI')
e = circuit.add([
    (gc, 'input'),
    (gc, 'output'),
    (y, 'splitter'),
    (y, 'recombiner'),
    (wg150, 'wg_long'),
    (wg50, 'wg_short'),
])

# You can set pin names individually:
circuit.elements['input'].pins['n2'] = 'input'
circuit.elements['output'].pins['n2'] = 'output'

# Or you can rename all the pins simultaneously:
circuit.elements['splitter'].pins = ('in1', 'out1', 'out2')
circuit.elements['recombiner'].pins = ('out1', 'in2', 'in1')

# Circuits can be connected using the elements' string names:
circuit.connect_many([
    ('input', 'n1', 'splitter', 'in1'),
    ('splitter', 'out1', 'wg_long', 'n1'),
    ('splitter', 'out2', 'wg_short', 'n1'),
    ('recombiner', 'in1', 'wg_long', 'n2'),
    ('recombiner', 'in2', 'wg_short', 'n2'),
    ('output', 'n1', 'recombiner', 'out1'),
])

# or by using the actual object reference.
# circuit.connect(e[0], e[0].pin[0], e[2], e[2].pin[0])

# At this point, your circuit is defined. This file can serve as a description
# for a subcircuit that is used in a larger circuit, and can simply be imported
# using the Python import system (`from mzi import circuit`).

# Run a simulation on the netlist.
simulation = SweepSimulation(circuit, 1500e-9, 1600e-9)
result = simulation.simulate()

f, s = result.data('input', 'output')
plt.plot(f, s)
plt.title("MZI")
plt.tight_layout()
plt.show()

# We can run a monte carlo simulation on the circuit, too
simulation = MonteCarloSweepSimulation(circuit, 1500e-9, 1600e-9)
runs = 10
result = simulation.simulate(runs=runs)

for i in range(1, runs + 1):
    f, s = result.data('input', 'output', i)
    plt.plot(f, s)

# The data located at the 0 position is the ideal values.
f, s = result.data('input', 'output', 0)
Add-Drop Filters

Note: This tutorial requires the installation of SiPANN. See Companion Libraries for more details.

In this tutorial, we’re going to create a simulation that would predict the frequency response of a single-input, multiple-output Add-Drop Filter.

An add-drop filter uses rings of different radii to “select” out specific frequencies from a main data line and convey them to an output.

Note: You should be familiar with the processes explained in the MZI Tutorial before doing this tutorial, as it provides a more detailed overview of decomposing circuits into blocks, declaring the models required in a circuit, and making connections between elements.

Code Walkthrough

This example walks through the file “filters.py”.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright © Simphony Project Contributors
# Licensed under the terms of the MIT License
# (see simphony/__init__.py for details)
#
# File: filters.py
```

For this tutorial, we will be using matplotlib and numpy to manipulate and view the results of our simulation.
We’ll need the following modules and objects from simphony:

- **sipann**: The SiPANN model library, which provides models trained using machine learning techniques. Since they’re not premodeled using a process such as FDTD, we’re not limited to specific parameterized devices for which simulations have already been performed.

- **simphony.library.siepic**: We use waveguide and terminator models from the siepic library.

- **simphony.netlist.Subcircuit**: We use the Subcircuit object to define our photonic circuits.

- **simphony.simulation.SweepSimulation**: The SweepSimulation takes our circuit and calculates the outputs for a frequency sweep simulation.

- **simphony.tools.freq2wl**: A convenience function provided by Simphony for converting frequencies to wavelengths.

The tutorial began with a block-diagram model of our final construction. Note the main input data line and terminators. We can declare the models we’ll use for those circuit instances.

```python
# Have a main data line where frequency multiplexed data enters the circuit.
wg_data = siepic.ebeam_wg_integral_1550(100e-6)

# A terminator for dispersing unused light
term = siepic.ebeam_terminator_te1550()
```

Our final product has a component that is duplicated three times with varying parameters. This kind of redundancy makes an excellent case for the use of the **factory method design pattern**.

`Fig. 3: A block-diagram model of a ring resonator alone. Note the port with a termination; light is never designed to travel in that direction, so any potential back-scattered light is simply dispersed.`

The following function generates a subcircuit of a ring resonator, as pictured above, with a radius given as a parameter. The subcircuit it returns can be used within another circuit, just like any regular, base model.

```python
def ring_factory(radius):
    
    Creates a full ring (with terminator) from a half ring.
    
    Ports of a half ring are ordered like so:
```
Resulting pins are ('in', 'out', 'pass').

Parameters
----------
radius: float
    The radius of the ring resonator, in nanometers.

```python
# Have rings for selecting out frequencies from the data line.
# See SiPANN's model API for argument order and units.
half_ring = SimphonyWrapper(spee.HalfRing(500, 220, radius, 100))

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

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

circuit.connect_many(
    ('input', 'midb', 'output', 'midb'),
    ('input', 'midt', 'output', 'midt'),
    ('terminator', 'nl', 'output', 'term')
)

return circuit
```

Before we construct the full add-drop filter, we can run a simulation on a single ring to make sure our code is behaving the way we’d expect.

```python
# Behold, we can run a simulation on a single ring resonator.
cir1 = ring_factory(10000)
sim1 = SweepSimulation(cir1, 1500e-9, 1600e-9)
resl = sim1.simulate()

f1, s = resl.data(resl.pinlist['in'], resl.pinlist['pass'])
plt.plot(f1, s)
plt.title("10-micron Ring Resonator")
plt.tight_layout()
plt.show()
```

Now we’ll add several of these ring resonators to our circuit. They will be cascaded together to create our filter.

```python
# Now, we’ll create the circuit (using several ring resonator subcircuits)
# and add all individual instances.
circuit = Subcircuit('Add-Drop Filter')
e = circuit.add(
    (continues on next page)
)```
Fig. 4: The through-port frequency response of a 10 micron ring resonator.
We can rename pins, as convenient, either individually or simulatedly. For ease of accessing outputs post-simulation, we'll rename some of the ports. Renaming requires prior knowledge of how ports are laid out on the device. For pin ordering on SiPANN models, see their documentation.

```python
# You can set pin names individually (here I'm naming all the outputs that # I'll want to access after the simulation has been run):
circuit.elements['input'].pins['n1'] = 'input'
circuit.elements['out1'].pins['n2'] = 'out1'
circuit.elements['out2'].pins['n2'] = 'out2'
circuit.elements['out3'].pins['n2'] = 'out3'
```

Now we'll define all circuit connections:

```python
circuit.connect_many([  
    ('input', 'n2', 'ring10', 'in'),  
    ('out1', 'n1', 'ring10', 'out'),  
    ('connect1', 'n1', 'ring10', 'pass'),  
    ('connect1', 'n2', 'ring11', 'in'),  
    ('out2', 'n1', 'ring11', 'out'),  
    ('connect2', 'n1', 'ring11', 'pass'),  
    ('connect2', 'n2', 'ring12', 'in'),  
    ('out3', 'n1', 'ring12', 'out'),  
    ('terminator', 'n1', 'ring12', 'pass'),  
])
```

Finally, let's run a sweep simulation. (Notice the reduced frequency range, since I'm interested in focusing in on only a few peaks, instead of a perhaps standard, full 1500nm-1600nm sweep.)

```python
# Run a simulation on the netlist.
simulation = SweepSimulation(circuit, 1524.5e-9, 1551.15e-9)
result = simulation.simulate()
```

The rest of this confusing “gridspec” code is to create a pretty plot that looks at the full sweep range and also a single peak that I'm particularly interested in. The main takeaway from this section is that getting the data out of a simulation object is as simple as calling `data` and providing the names of the pins you’re using as an input and output.

```python
fig = plt.figure(tight_layout=True)
gs = gridspec.GridSpec(1, 3)
```
```python
from matplotlib.pyplot import * 
import matplotlib.gridspec as gridspec 
import numpy as np 
from SiPANN.scee_int import SimphonyWrapper 
from simphony.library import siepic 
from simphony.netlist import Subcircuit 
from simphony.simulation import SweepSimulation 
from simphony.tools import freq2wl 

# Have a main data line where frequency multiplexed data enters the circuit. 
wg_data = siepic.ebeam_wg_integral_1550(100e-6) 

# A terminator for dispersing unused light 
term = siepic.ebeam_terminator_te1550() 
```

(continues on next page)
Fig. 5: The response of our designed add-drop filter.
def ring_factory(radius):
    """
    Creates a full ring (with terminator) from a half ring.

    Ports of a half ring are ordered like so:
    2    4
    |    |
    \   /  \\
    \ /   / ---========---
    1    3

    Resulting pins are ('in', 'out', 'pass').
    """
    # Have rings for selecting out frequencies from the data line.
    # See SiPANN's model API for argument order and units.
    half_ring = SimphonyWrapper(scee.HalfRing(500, 220, radius, 100))

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

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

    circuit.connect_many([
        ('input', 'midb', 'output', 'midb'),
        ('input', 'midt', 'output', 'midt'),
        ('terminator', 'nl', 'output', 'term')
    ])  
    return circuit

    # Behold, we can run a simulation on a single ring resonator.
    cir1 = ring_factory(10000)
    sim1 = SweepSimulation(cir1, 1500e-9, 1600e-9)
    res1 = sim1.simulate()
    f1, s = res1.data(res1.pinlist['in'], res1.pinlist['pass'])
    plt.plot(f1, s)
    plt.title("10-micron Ring Resonator")
    plt.tight_layout()
    plt.show()

    # Now, we’ll create the circuit (using several ring resonator subcircuits)
    # and add all individual instances.
circuit = Subcircuit('Add-Drop Filter')

e = circuit.add([
    (wg_data, 'input'),
    (ring_factory(10000), 'ring10'),
    (wg_data, 'out1'),

    (wg_data, 'connect1'),
    (ring_factory(11000), 'ring11'),
    (wg_data, 'out2'),

    (wg_data, 'connect2'),
    (ring_factory(12000), 'ring12'),
    (wg_data, 'out3'),

    (term, 'terminator')
])

# You can set pin names individually (here I'm naming all the outputs that
# I'll want to access after the simulation has been run):

circuit.elements['input'].pins['n1'] = 'input'
circuit.elements['out1'].pins['n2'] = 'out1'
circuit.elements['out2'].pins['n2'] = 'out2'
circuit.elements['out3'].pins['n2'] = 'out3'

circuit.connect_many([
    ('input', 'n2', 'ring10', 'in'),
    ('out1', 'n1', 'ring10', 'out'),
    ('connect1', 'n1', 'ring10', 'pass'),

    ('connect1', 'n2', 'ring11', 'in'),
    ('out2', 'n1', 'ring11', 'out'),
    ('connect2', 'n1', 'ring11', 'pass'),

    ('connect2', 'n2', 'ring12', 'in'),
    ('out3', 'n1', 'ring12', 'out'),
    ('terminator', 'n1', 'ring12', 'pass'),
])

# Run a simulation on the netlist.
simulation = SweepSimulation(circuit, 1524.5e-9, 1551.15e-9)
result = simulation.simulate()

fig = plt.figure(tight_layout=True)
gs = gridspec.GridSpec(1, 3)

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

f, s = result.data('input', 'out1')
ax.plot(freq2wl(f)*1e9, s, label='Output 1', lw='0.7')
f, s = result.data('input', 'out2')
ax.plot(freq2wl(f)*1e9, s, label='Output 2', lw='0.7')
f, s = result.data('input', 'out3')
ax.plot(freq2wl(f)*1e9, s, label='Output 3', lw='0.7')

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

(continues on next page)
The Green Machine

Code Walkthrough

This example walks through the file “gm.py”.

It will be completed when I get enough time to do a write-up for it. For now, reference the first two available examples. The code for this simulation is still available, though.

Full Code Listing

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
#
# Copyright © Simphony Project Contributors
# Licensed under the terms of the MIT License
# (see simphony/__init__.py for details)

import os
import sys
sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
import matplotlib.pyplot as plt
import numpy as np
from simphony.library import ebeam, sipann
from simphony.netlist import Subcircuit
from simphony.simulation import SweepSimulation
from simphony.tools import freq2wl, wl2freq

# We can rename the pins attribute on the class before we instantiate them;
# then we don’t have to rename the pins on each element individually later.
ebeam.ebeam_wg_integral_1550.pins = ('in', 'out')
sipann.sipann_dc_fifty.pins = ('in1', 'in2', 'out1', 'out2')
sipann.sipann_dc_crossover1550.pins = ('in1', 'in2', 'out1', 'out2')

# Get all the models we’re going to need for the green machine circuit:
 gc = ebeam.ebeam_gc_te1550()
```

(continues on next page)
wg100 = ebeam.ebeam_wg_integral_1550(length=100e-6)
dc = sipann.sipann_dc_fifty()
crossover = sipann.sipann_dc_crossover1550()
wgin2 = ebeam.ebeam_wg_integral_1550(length=102.125e-6)
wg300 = ebeam.ebeam_wg_integral_1550(length=300e-6)

# Add all the elements used in the circuit
circuit = Subcircuit('Green Machine')
e = circuit.add(
    # Define the four input grating couplers
    (gc, 'in1'),
    (gc, 'in2'),
    (gc, 'in3'),
    (gc, 'in4'),

    # The grating couplers each feed into their own waveguide
    (wg100, 'wg1'),
    (wg100, 'wg2'),
    (wg100, 'wg3'),
    (wg100, 'wg4'),

    # Each pair of waveguides feeds into a 50/50 directional coupler
    (dc, 'dc1'),
    (dc, 'dc2'),

    # After mixing, the center pair of waveguides cross paths at a 100/0 crossing. The edge pair of waveguides pass uninterrupted.
    (wg300, 'wg_pass1'),
    (wg100, 'wg_in1'), (wgin2, 'wg_out1'),
    (crossover, 'crossing'),
    (wg100, 'wg_in2'), (wgin2, 'wg_out2'),
    (wg300, 'wg_pass2'),

    # After crossing, the waveguides are mixed again.
    (dc, 'dc3'),
    (dc, 'dc4'),

    # The outputs are fed through waveguides.
    (wg100, 'wg5'),
    (wg100, 'wg6'),
    (wg100, 'wg7'),
    (wg100, 'wg8'),

    # We finally output the values through grating couplers.
    (gc, 'out1'),
    (gc, 'out2'),
    (gc, 'out3'),
    (gc, 'out4'),
)

# Let's rename some ports on some of our elements so that we can:
# 1) find them again later, and
# 2) make our code clearer by using plain english for the connections.
circuit.elements['in1'].pins['nl'] = 'in1'
circuit.elements['in2'].pins['nl'] = 'in2'
circuit.elements['in3'].pins['nl'] = 'in3'
circuit.elements['in4'].pins['nl'] = 'in4'

(continues on next page)

2.1. Tutorials
circuit.elements['out1'].pins['n2'] = 'out1'
circuit.elements['out2'].pins['n2'] = 'out2'
circuit.elements['out3'].pins['n2'] = 'out3'
circuit.elements['out4'].pins['n2'] = 'out4'

# Phew! Now that we got all those elements out of the way, we can finally
# work on the circuit connections.
circuit.connect_many(
    ('in1', 'n2', 'wg1', 'in'), ('in2', 'n2', 'wg2', 'in'), ('in3', 'n2', 'wg3', 'in'), ('in4', 'n2', 'wg4', 'in'),
    ('wg1', 'out', 'dc1', 'in1'), ('wg2', 'out', 'dc1', 'in2'), ('wg3', 'out', 'dc2', 'in1'), ('wg4', 'out', 'dc2', 'in2'),
    ('dc1', 'out1', 'wg_pass1', 'in'), ('dc2', 'out1', 'wg_pass1', 'in'), ('dc1', 'out2', 'wg_in1', 'in'), ('dc2', 'out2', 'wg_in1', 'in'), ('wg_in1', 'out', 'crossing', 'in1'), ('wg_in2', 'out', 'crossing', 'in2'), ('crossing', 'out1', 'wg_out1', 'in'), ('crossing', 'out2', 'wg_out2', 'in'),
    ('wg_pass1', 'out', 'dc3', 'in1'), ('wg_pass2', 'out', 'dc3', 'in2'), ('wg_out1', 'out', 'dc3', 'in1'), ('wg_out2', 'out', 'dc3', 'in2'), ('dc3', 'out1', 'wg5', 'in'), ('dc3', 'out2', 'wg6', 'in'), ('dc4', 'out1', 'wg7', 'in'), ('dc4', 'out2', 'wg8', 'in'),
    ('wg5', 'out', 'out1', 'n1'), ('wg6', 'out', 'out2', 'n1'), ('wg7', 'out', 'out3', 'n1'), ('wg8', 'out', 'out4', 'n1'),
)

# Run a simulation on our circuit.
simulation = SweepSimulation(circuit, 1549.9e-9, 1550.1e-9)
result = simulation.simulate()

# Get the simulation results
f, s = result.data(result.pinlist['in1'], result.pinlist['out1'])

# The Green Machine is optimized for 1550 nanometers. We'd like to investigate
# its behavior at that specific frequency:
set_freq = w12freq(1550e-9)
in_port = 'in1'
plt.figure()
plt.plot(*result.data(result.pinlist[in_port], result.pinlist['out1']), label='1 to 5
   →')
plt.plot(*result.data(result.pinlist[in_port], result.pinlist['out2']), label='1 to 6 →')
plt.plot(*result.data(result.pinlist[in_port], result.pinlist['out3']), label='1 to 7 →')
plt.plot(*result.data(result.pinlist[in_port], result.pinlist['out4']), label='1 to 8 →')
plt.axvline(set_freq)
plt.legend(loc="upper right")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Fractional Optical Power")
plt.show()

# We're interested now in the phase offsets at our wavelength of interest.
plt.figure()
freq, s = result.f, result.s
idx = np.argmax(freq>set_freq)
input_pin = result.pinlist['in1'].index
outputs = [result.pinlist['out' + str(n)].index for n in range(1,5)]
offset = min(np.angle(s[idx, outputs, input_pin]))
# angles = np.unwrap(np.angle(s[:, outputs, input_pin])).T - offset
angles = np.angle(s[:, outputs, input_pin]).T - offset
for angle in angles:
    plt.plot(freq2wl(freq)*1e9, angle, linewidth='0.7')
plt.axvline(1550, color='k', linestyle='--', linewidth='0.5')
plt.legend([r'$\phi_4$',r'$\phi_5$',r'$\phi_6$',r'$\phi_7$'], loc='upper right')
plt.xlabel("Wavelength (nm)")
plt.ylabel("Phase")
plt.show()

import sys
sys.exit()

plt.figure()
idx = np.argmax(freq>set_freq)
print(idx, freq2wl(freq[idx]))
angles = np.rad2deg(np.unwrap(np.angle(s[:,outputs,input_pin]))).T
angles = angles + ((angles[:,idx] % 2*np.pi) - angles[:,idx]).reshape((4,1))
print(angles[:,idx], angles)
for i in range(4):
    plt.plot(freq2wl(freq)*1e9, angles[i], label="Port {} to {}".format(i, j))
    plt.plot(freq2wl(freq[idx])*1e9, angles[i][idx], 'rx')
plt.axvline(1550)
# plt.legend()
plt.xlabel("Wavelength (nm)")
plt.ylabel("Phase")
plt.show()

import sys
sys.exit()
# plt.axvline(set_freq/1e12)
plt.show()
# Response at precisely 1550nm
#
idx = np.argmax(freq>set_freq)
print(idx, freq2wl(freq[idx]))

# Phases of the four outputs at 1550nm
plt.figure()
circle = np.linspace(0, 2*np.pi)
plt.plot(np.cos(circle), np.sin(circle))

# for i in range(0,4):
inputs1550 = [0] * 8
for output in range(4,8):
    rad = np.angle(s[idx, output, i])
    plt.plot(np.cos(rad), np.sin(rad), 'o')
    inputs1550[output-4] = np.cos(rad) + np.sin(rad) * 1j
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.axes().set_aspect('equal')

# for val in inputs1550:
#    print(val, np.rad2deg(np.angle(val)))

# Multiple input stuffs:

# def multi_input(num_ports, inputs, verbose=True):
#     inputs = np.array(inputs, dtype=np.complex_)
#     if verbose:
#         angles = np.rad2deg(np.angle(inputs))
#         print(angles - min(angles))
#     out = np.zeros([len(freq), num_ports], dtype='complex128')
#     for j in range(len(freq)):
#         out[j, :] = np.dot(s[j, :, :], inputs.T)
#     return out

def plot_outputs(out):
    plt.figure()
    for j in range(8):
        plt.subplot(8, 1, j+1)
        plt.plot(freq/1e12, np.abs(out[:, j])**2, label="Port {0}".format(j))
        plt.axvline(set_freq/1e12)
        plt.legend()
        plt.xlabel("Frequency (THz)")
        plt.ylabel("Normalized Power")
out = multi_input(8, inputs1550)
plt.figure()
for j in range(8):
    plt.subplot(8, 1, j+1)
    plt.plot(freq/1e12, np.abs(out[:, j])**2, label="Port {0}".format(j))
    plt.axvline(set_freq/1e12)
    plt.legend()
    plt.xlabel("Frequency (THz)")
    plt.ylabel("Normalized Power")

(continues on next page)
Simphony includes tools for simulating circuits designed in KLayout using the SiEPIC EBeam Process Design Kit (PDK).

2.1.2 Creating Model Libraries

API for Implementing Models

This is how we implement model libraries.

```python
class simphony.elements.Model
    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 the attributes of an element (see Attributes).
    Following the general EAFP coding style of Python, errors will only be raised when an unimplemented function
    is called, not when the class instance is created.

    pins
        A tuple of all the default pin names of the device. Length of default tuple should be equal to the number
        of ports on the device.

        Type: tuple of str

    freq_range
        A tuple of the valid frequency bounds for the element in the order (lower, upper). Can be made (-infty, infty)
        be setting to (None, None).

        Type: tuple of float

    monte_carlo_s_parameters(freq)
        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
            freq (np.ndarray) -- 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

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(freq)

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

Parameters freq (np.ndarray) – 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.

2.2 Simphony Techniques

These documents are intended as recipes for common tasks using Simphony. For detailed reference documentation of
the functions and classes contained in the package, see the API reference.

2.2.1 The Factory Method Design Pattern

Prerequisites

Before reading this tutorial you should already be familiar with the basic objects and models used by Simphony.
Knowledge of how SPICE models work in many electronics simulation software packages may also be helpful.

Learner profile

This tutorial is intended as a quick overview of the factory pattern and how it can be employed to help you create
robust, easy-to-customize, large-scale circuits without hard-coding every detail.

Learning Objectives

After this tutorial, you should be able to:
• Identify situations where this design pattern should be applied to avoid code repetition when creating large circuits.
• Write your own factory method two produce parameterized subcircuits.
• Cascade subcircuits together in a loop to create larger circuits.

The Factory Method

What is the factory pattern?

The factory design pattern is a design pattern where there exists several related types of objects and a “factory” generates desired objects based on parameters received or attributes desired. 

For our purposes, the factory pattern becomes appealing anytime we want to create several parameterized variations of a compound structure using simple elements (or even other compound elements, as they can be nested).

What kind of structures warrant the factory pattern?

Here’s a few examples of when a factory pattern would be useful:

• A circuit composed of MZI’s of varying length.
• A circuit composed of ring resonators with radii spread over some range.
• A circuit composed of bragg grating structures with different periodicities.

2.3 Model Libraries

Simphony comes with several model libraries included that can be used to construct circuit simulations. 

Simphony includes two default component libraries. Components from those libraries and their usage is included below.

2.3.1 EBeam model library

class simphony.library.ebeam.ebeam_bdc_te1550
    Bases: simphony.elements.Model

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.

```
 freq_range = (187370000000000.0, 199862000000000.0)
```

The valid frequency range for this model.
\texttt{monte\_carlo\_s\_parameters}(freq)

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 \texttt{s\_parameters}().

\textbf{Parameters} \texttt{freq}(\texttt{np.ndarray}) – The frequency range to generate monte carlo s-parameters over.

\textbf{Returns} \texttt{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 \texttt{monte\_carlo\_s\_parameters} would be (1, 4, 4).

\textbf{Return type} \texttt{np.ndarray}

\texttt{pins} = ('n1', 'n2', 'n3', 'n4')

The default pin names of the device

\texttt{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 \texttt{MonteCarloSweepSimulation} calls this function once per run over the circuit.

\textbf{Notes}

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

\texttt{s\_parameters}(freq)

Returns scattering parameters for the element with its given parameters as declared in the optional \texttt{\_init\_}().

\textbf{Parameters} \texttt{freq}(\texttt{np.ndarray}) – The frequency range to get scattering parameters for.

\textbf{Returns} \texttt{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 \texttt{s\_parameters} would be (1, 4, 4).

\textbf{Return type} \texttt{np.ndarray}

\textbf{Raises} \texttt{NotImplementedError} – Raised if the subclassing element doesn’t implement this function.

\texttt{class} \texttt{simphony.library.ebeam.ebeam\_dc\_halfring\_te1550}

\texttt{Bases: simphony.elements.Model}

\texttt{freq\_range} = (187370286250000.0, 19986163866666.66)

The valid frequency range for this model.

\texttt{monte\_carlo\_s\_parameters}(freq)

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 \texttt{s\_parameters}().

\textbf{Parameters} \texttt{freq}(\texttt{np.ndarray}) – 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

```python
pins = ('n1', 'n2')
```

The default pin names of the device.

```python
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.

```python
s_parameters(freq)
```

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

Parameters  

- `freq` *(np.ndarray)* – 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.

```python
class simphony.library.ebeam.ebeam_dc_te1550
```

Bases: `simphony.elements.Model`

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.

```python
freq_range = (187370000000000.0, 199862000000000.0)
```

The valid frequency range for this model.
**monte_carlo_s_parameters**(freq)

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**

- `freq (np.ndarray)` - 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`

**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**(freq)

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

**Parameters**

- `freq (np.ndarray)` - 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.library.ebeam.ebeam_gc_te1550

**Bases:** simphony.elements.Model

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].

---

40 Chapter 2. Using Simphony
freq_range = (176348504705882.4, 214137470000000.0)
The valid frequency range for this model.

monte_carlo_s_parameters (freq)
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 freq (np.ndarray) – 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

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

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(freq)`

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

**Parameters**
- `freq` *(np.ndarray)* – 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.library.ebeam.ebeam_terminator_te1550

**Bases:** simphony.elements.Model

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.

```python
freq_range = (181692000000000.0, 206753000000000.0)
```

The valid frequency range for this model.

```python
monte_carlo_s_parameters(freq)
```

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**
- `freq` *(np.ndarray)* – 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

```python
pins = ('n1',)
```

The default pin names of the device

```python
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 \texttt{MonteCarloSweepSimulation} calls this function once per run over the circuit.

\section*{Notes}
This function should not accept any parameters, but may act on instance or class attributes.

\begin{verbatim}
s_parameters(freq)
    Returns scattering parameters for the element with its given parameters as declared in the optional \_\_init\_\_().

    Parameters freq (np.ndarray) -- 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 \texttt{NotImplementedError} -- Raised if the subclassing element doesn't implement this function.
\end{verbatim}

\begin{verbatim}
class simphony.library.ebeam.ebeam_wg_integral_1550(length, lam0=1.55e-06, ne=2.44553, ng=4.19088, nd=0.000354275, sigma_ne=0.05, sigma_ng=0.05, sigma_nd=0.0001)

Bases: simphony.elements.Model

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

A waveguide easily connects other optical components within a circuit.
\end{verbatim}
- **sigma_nd** *(float, optional)* – Standard deviation of the group dispersion (default 0.0001).

**Notes**

The \texttt{sigma_} values in the parameters are used for monte carlo simulations.

\texttt{freq\_range = (187370000000000.0, 199862000000000.0)}

The valid frequency range for this model.

\texttt{monte\_carlo\_s\_parameters(freq)}

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

In this implementation of the monte carlo routine, random values are generated for \texttt{ne}, \texttt{ng}, and \texttt{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) \texttt{ne}, \texttt{ng}, and \texttt{nd} values. Hence, there is correlated randomness in the monte carlo parameters but they are consistent within a single circuit.

\texttt{pins = ('n1', 'n2')}

The default pin names of the device.

\texttt{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 \texttt{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.

\texttt{s\_parameters(freq)}

Get the s-parameters of a waveguide.

**Parameters**

- **start** *(float)* – The starting frequency to obtain s-parameters for (in Hz).
- **stop** *(float)* – The ending frequency to obtain s-parameters for (in Hz).
- **num** *(int)* – The number of points to use between start\_freq and stop\_freq.

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

**Return type** tuple

\texttt{class simphony.library.ebeam.ebeam\_y\_1550}

\texttt{Bases: simphony.elements.Model}

The y-branch efficiently splits the input between the two outputs.
freq_range = (187370000000000.0, 199862000000000.0)
The valid frequency range for this model.

monte_carlo_s_parameters(freq)
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
freq (np.ndarray) – 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

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

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

Parameters
freq (np.ndarray) – 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.

### 2.3.2 SiEPIC model library

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.

```python
class simphony.library.siepic.ebeam_bdc_te1550 (thickness=2.2e-07, width=5e-07)
    Bases: simphony.library.siepic.siepic_ebeam_pdk_base

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.

```
• 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).

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

class simphony.library.siepic.ebeam_dc_te1550 (gap=2e-07, Lc=1e-05)
Bases: simphony.library.siepic.siepic_ebeam_pdk_base

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).

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

class simphony.library.siepic.ebeam_terminator_te1550 (w1=5e-07, w2=6e-08, L=1e-05)
Bases: simphony.library.siepic.siepic_ebeam_pdk_base

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).
pins = ('n1',)
The default pin names of the device

class simphony.library.siepic.ebeam_gc_te1550(thickness=2.2e-07, deltax=0, polarization='TE')
Bases: simphony.library.siepic.siepic_ebeam_pdk_base

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’.

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

class simphony.library.siepic.ebeam_wg_integral_1550(length=0.0, width=5e-07, height=2.2e-07, polarization='TE', sigma_ne=0.05, sigma_ng=0.05, sigma_nd=0.0001)
Bases: simphony.library.siepic.siepic_ebeam_pdk_base

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.

```python
pins = ('n1', 'n2')
```

The default pin names of the device

class simphony.library.siepic.ebeam_y_1550 (thickness=2.2e-07, width=5e-07, polarization='TE')

Bases: simphony.library.siepic.siepic_ebeam_pdk_base

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’.

2.3. Model Libraries 49
pins = ('n1', 'n2', 'n3')
The default pin names of the device

2.3.3 SiPANN model library

A separate Python program exists with a model library built around Simphony. This program is called SiPANN (Silicon Photonics with Artificial Neural Networks). It leverages various machine learning techniques to simulate integrated photonic device circuits, meaning it can very quickly and accurately simulate devices with varying parameters (such as waveguide width or thickness) without having to run a full, slow, FDTD simulation before a designed device can be used in a photonic integrated circuit (PIC) design software such as Simphony.

Installation instructions and documentation, including the available model library and its API, can be found in SiPANN’s Documentation.

2.4 Simulators

2.4.1 Sweep Simulation

class simphony.simulation.SweepSimulation(circuit: simphony.netlist.Subcircuit, start: float = 1.5e-06, stop: float = 1.6e-06, num: int = 2000, mode='wl')

A swept simulation.

Parameters

• circuit (Subcircuit) – The circuit to be simulated.

• start (float) – The start wavelength (in meters) or frequency (in Hz).

• stop (float) – The stop wavelength (in meters) or frequency (in Hz).

• num (int, optional) – The number of sampled points.

• mode (str, optional) – Defines sweep range mode; either ‘wl’ for wavelength (m) or ‘freq’ for frequency (Hz).

freq

The frequency array over which the simulation is performed.

Type np.ndarray

static connect_circuit (netlist)

Connects the s-matrices of a photonic circuit given its Netlist and returns a single ‘SimulatedComponent’ object containing the frequency array, the assembled s-matrix, and a list of the external nets (negative integers).

Parameters

• component_list (List[SimulatedComponent]) – A list of the components to be connected.

• net_count (int) – The total number of internal nets in the component list.

Returns combined – After the circuit has been fully connected, the result is a single ComponentSimulation with fields f (frequency), s (s-matrix), and nets (external ports: negative numbers, as strings).

Return type ScatteringMatrix
Notes

This function doesn’t actually store combined on each iteration through the netlist. That’s because the Pin objects can only reference one PinList at a time, which in turn can only reference one Element. Since we transferring the actual Pin objects between lists, keeping a reference to the Pin also keeps a reference to the combined Element alive. Hence, we track pins but not the SimulationResult.

simulate()
Runs the simulation on the object’s circuit.

    Returns sim – A loaded SweepSimulationResult object.
    Return type SweepSimulationResult

static validate_models(models, freq)
Ensures all models are valid over the specified frequency range.

Parameters

• models (list) – A list of the model objects to be verified.
• freq (np.ndarray) – The array of frequency values the simulation is defined over.

Raises

• NotImplementedError – If a model does not have a class attribute freq_range defining the valid frequency range for the model.
• ValueError – If the simulation frequencies are outside of the range of the valid frequencies for a model.

class simphony.simulation.SweepSimulationResult(freq, smat)
A simulation result for a swept simulation.

Parameters

• freq (np.array) – A numpy array of the frequency values in its simulation.
• smat (ScatteringMatrix) – A numpy array of the s-parameter matrix for the given frequency range.

data(inp, outp, dB=False)

Parameters

• inp (str or Pin) – Input pin.
• outp (str or Pin) – Output pin.

2.4.2 Monte Carlo Sweep Simulation

class simphony.simulation.MonteCarloSweepSimulation(circuit: simphony.netlist.Subcircuit, start: float = 1.5e-06, stop: float = 1.6e-06, num: int = 2000, mode='wl')
A monte carlo sweep simulation.

Parameters

• circuit (Subcircuit) – The circuit to be simulated.
• start (float) – The start wavelength (in meters) or frequency (in Hz).
• **stop** (*float*) – The stop wavelength (in meters) or frequency (in Hz).
• **num** (*int*) – The number of sampled points.
• **mode** (*str*) – Defines sweep range mode; either ‘wl’ for wavelength (m) or ‘freq’ for frequency (Hz).

**static connect_circuit** (*netlist*)
Connects the s-matrices of a photonic circuit given its Netlist and returns a single ‘SimulatedComponent’ object containing the frequency array, the assembled s-matrix, and a list of the external nets (negative integers).

**Parameters**

• **component_list** (*List[SimulatedComponent]*) – A list of the components to be connected.

• **net_count** (*int*) – The total number of internal nets in the component list.

**Returns combined** – After the circuit has been fully connected, the result is a single ComponentSimulation with fields f (frequency), s (s-matrix), and nets (external ports: negative numbers, as strings).

**Return type** ScatteringMatrix

**Notes**
This function doesn’t actually store combined on each iteration through the netlist. That’s because the Pin objects can only reference one PinList at a time, which in turn can only reference one Element. Since we transferring the actual Pin objects between lists, keeping a reference to the Pin also keeps a reference to the combined Element alive. Hence, we track pins but not the SimulationResult.

**simulate** (*runs=10*)

**Parameters**

• **runs** (*int, optional*) – The number of monte carlo iterations to run (default 10).

**static validate_models** (*models, freq*)
Ensures all models are valid over the specified frequency range.

**Parameters**

• **models** (*list*) – A list of the model objects to be verified.

• **freq** (*np.ndarray*) – The array of frequency values the simulation is defined over.

**Raises**

• **NotImplementedError** – If a model does not have a class attribute freq_range defining the valid frequency range for the model.

• **ValueError** – If the simulation frequencies are outside of the range of the valid frequencies for a model.

**class** *simphony.simulation.MonteCarloSimulationResult* (*freq, smat, runs*)

**Parameters**

• **freq** (*np.ndarray*) –

• **smat** (*simphony.simulation.ScatteringMatrix*) –

• **runs** (*int*) –

• **data** (*inp, outp, run, dB=False*)

**Chapter 2. Using Simphony**
Parameters

- \texttt{inp(str or Pin)} – Input pin.
- \texttt{outp(str or Pin)} – Output pin.

2.5 Simphony Integrations

Simphony integrates well with several other programs. Additionally, Simphony has modules that allow other software to integrate well into Simphony. These are described here.

If you have written an integration for Simphony that is publically available, please let us know so we can add it to this list!

2.5.1 SiEPIC Integration

Simphony initially began as an extension to [SiEPIC-Tools](https://github.com/lukasc-ubc/SiEPIC-Tools), but was officially born once it became large enough to be considered its own stand-alone project. There is a repository forked from lukase-ubc/SiEPIC-Tools, [SiEPIC-Tools](https://github/sequoiap/SiEPIC-Tools), that integrates Simphony with SiEPIC-Tools and KLayout in order to perform photonic circuit simulations using a layout-driven design methodology.
Simphony was developed by CamachoLab at Brigham Young University but also strives to be an open-source project that welcomes the efforts of volunteers. If there is anything you feel can be improved, functionally or in our documentation, we welcome your feedback – let us know what the problem is or open a pull request with a fix!

More information about the development of Simphony can be found at our project webpage. The documentation is hosted for free at https://simphonyphotonics.readthedocs.io/. The source for this documentation can be found in the master branch of the repository.

- Documenting The Simphony Project: A Guide to Simphony Documentation \ Building the Simphony API and reference docs
- Contributing to Simphony: Contributing to Simphony
- Bugs and Feature Requests: Report a Bug

### 3.1 Contributing to Simphony

Not a coder? Not a problem! Simphony is still a developing project, and we could use a lot of help. These are all activities we’d like to get help with (they’re all important, so we list them in alphabetical order):

- Code maintenance and development (architecture input welcome)
- Fundraising
- Marketing
- Writing technical documentation and examples

#### 3.1.1 Development process - summary

Here’s the short summary, complete TOC links are below:

1. If you are a first-time contributor:
   - Go to https://github.com/BYUCamachoLab/simphony and click the “fork” button to create your own copy of the project.
   - Clone the project to your local computer:

        ```bash
        git clone https://github.com/your-username/simphony.git
        ```

   - Change the directory:
cd symphony

• Add the upstream repository:

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

• Now, `git remote -v` will show two remote repositories named:
  
  - `upstream`, which refers to the `simphony` repository
  - `origin`, which refers to your personal fork

2. Develop your contribution:

• Pull the latest changes from upstream:

  ```
git checkout master
  git pull upstream master
  ```

• Create a branch for the feature you want to work on. Use a sensible, “human-readable” name such as “monte-carlo-simualtions”:

  ```
git checkout -b monte-carlo-simualtions
  ```

• Commit locally as you progress (`git add` and `git commit`). Use a clear and meaningful commit message, write tests that fail before your change and pass afterward, run all the tests locally. Be sure to document any changed behavior in docstrings, keeping to the NumPy docstring standard.

3. To submit your contribution:

• Push your changes back to your fork on GitHub:

  ```
git push origin monte-carlo-simualtions
  ```

• Enter your GitHub username and password (repeat contributors or advanced users can remove this step by connecting to GitHub with SSH).

• Go to GitHub. The new branch will show up with a green Pull Request button. Make sure the title and message are clear, concise, and self-explanatory. Then click the button to submit it.

4. Review process:

• Reviewers (the other developers and interested community members) will write inline and/or general comments on your Pull Request (PR) to help you improve its implementation, documentation and style. Every single developer working on the project has their code reviewed, and we’ve come to see it as friendly conversation from which we all learn and the overall code quality benefits. Therefore, please don’t let the review discourage you from contributing: its only aim is to improve the quality of project, not to criticize (we are, after all, very grateful for the time you’re donating!).

• To update your PR, make your changes on your local repository, commit, run tests, and only if they succeed push to your fork. As soon as those changes are pushed up (to the same branch as before) the PR will update automatically. If you have no idea how to fix the test failures, you may push your changes anyway and ask for help in a PR comment.

• Various continuous integration (CI) services are triggered after each PR update to build the code, run unit tests, measure code coverage and check coding style of your branch. The CI tests must pass before your PR can be merged. If CI fails, you can find out why by clicking on the “failed” icon (red cross) and inspecting the build and test log. To avoid overuse and waste of this resource, test your work locally before committing.
• A PR must be **approved** by at least one core team member before merging. Approval means the core team member has carefully reviewed the changes, and the PR is ready for merging.

5. Document changes

Beyond changes to a functions docstring and possible description in the general documentation, if your change introduces any user-facing modifications they may need to be mentioned in the release notes. To ensure your change gets added to the release notes, be sure to mention what is changing in your pull request.

If your change introduces a deprecation, also make sure to include this fact in the pull request.

6. Cross referencing issues

If the PR relates to any issues, you can add the text `xref gh-xxxx` where `xxxx` is the number of the issue to github comments. Likewise, if the PR solves an issue, replace the `xref` with `closes`, `fixes` or any of the other flavors `github` accepts.

For a more detailed discussion, read on and follow the links at the bottom of this page.

**Divergence between upstream/master and your feature branch**

If GitHub indicates that the branch of your Pull Request can no longer be merged automatically, you have to incorporate changes that have been made since you started into your branch. Our recommended way to do this is to rebase on master.

**Guidelines**

• All code should have tests (see **test coverage** below for more details).

• All code should be **documented**.

**Stylistic Guidelines**

• Set up your editor to follow PEP 8 (remove trailing white space, no tabs, etc.). Check code with pyflakes / flake8. .. FIXME: Do we want to use **black** instead?

**Test coverage**

Pull requests (PRs) that modify code should either have new tests, or modify existing tests to fail before the PR and pass afterwards. You should **run the tests** before pushing a PR.

Running Simphony’s test suite locally requires some additional packages, such as pytest. The additional testing dependencies are listed in `test_requirements.txt` in the top-level directory, and can conveniently be installed with:

```
pip install -r test_requirements.txt
```

Tests for a module should ideally cover all code in that module, i.e., statement coverage should be at 100%.

To measure the test coverage, install `pytest-cov` and then run:

```
$ pytest --cov=simphony tests/
```
Building docs

To build docs, run `make` from the `docs` directory. `make help` lists all targets. For example, to build the HTML documentation, you can run:

```
make html
```

Then, all the HTML files will be generated in `docs/build/html/`. Since the documentation is based on docstrings, the appropriate version of Simphony must be installed in the host python used to run sphinx.

Requirements

Sphinx is needed to build the documentation.

These additional dependencies for building the documentation are listed in `doc_requirements.txt` and can be conveniently installed with:

```
ip install -r doc_requirements.txt
```

The documentation includes mathematical formulae with LaTeX formatting. A working LaTeX document production system (e.g. texlive) is required for the proper rendering of the LaTeX math in the documentation.

3.1.2 Development process - details

Setting up and using your development environment

Recommended development setup

For this chapter we assume that you have already set up your git repo.

Testing builds

Before submitting any pull requests, however, you should ensure that a pip installation of your updated package installs and functions properly. To test this, try installing your package locally by removing all installed versions of Simphony (by running `pip3 uninstall simphony` repeatedly until no installations remain) and running the following commands (from Simphony’s toplevel directory):

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

Note: Remember that all tests of Simphony should pass before committing your changes.

Using `pytest` is the recommended approach to running tests (see Running tests).
**Building in-place**

For development, you can set up an in-place build so that changes made to `.py` files have effect without rebuild. First, run:

```
$ pip install -e .
```

This allows you to import the in-place built Simphony from any location the Python environment used to install it is activated. If this is your system Python, you will be able to use the Simphony package from any directory.

Now editing a Python source file in Simphony allows you to immediately test and use your changes (in `.py` files), without even restarting the interpreter.

**Using virtual environments**

A frequently asked question is “How do I set up a development version of NumPy in parallel to a released version that I use to do my job/research?”.

One simple way to set up a development version of Simphony in parallel with a regular install is to install the released version in site-packages (perhaps using pip) and set up the development version in a virtual environment. The `venv` module should be included as part of a standard Python 3 installation. Create your virtualenv (named simphony-dev here) with:

```
$ python3 -m venv simphony-dev
```

Now, whenever you want to switch to the virtual environment, you can use the command `source simphony-dev/bin/activate` in the directory you created the virtual environment in, and `deactivate` to exit from the virtual environment back to your previous shell.

**Running tests**

Any code that you have contributed should also have accompanying tests. The style we adhere to in Simphony is to include a `/tests` directory wherever you are developing your module and writing tests using the `pytest` framework.

To install testing dependencies, run:

```
$ pip install .[test]
```

To run all tests, simply execute:

```
$ pytest
```

from the toplevel directory. Note that pytest must be run under a Python 3 environment.

Running individual test files can be useful; it can be much faster than running the whole test suite. This can be done with:

```
$ pytest path_to_testfile/test_file.py
```
Rebuilding & cleaning the workspace

There is no need to rebuilding Simphony after making changes to code if the installation procedure discussed in Building in-place is followed. After/while making changes, however, you may want to clean the workspace. The standard way of doing this is *(note: deletes any uncommitted files!)*:

```bash
$ git clean -xdf
```

When you want to discard all changes and go back to the last commit in the repo, use one of:

```bash
$ git checkout .
$ git reset --hard
```

Development workflow

**Prerequisites:** You already have your own forked copy of the Simphony repository, you have configured git, and have linked the upstream repository.

What is described below is a recommended workflow with Git.

**Basic workflow**

In short:

1. Start a new *feature branch* for each set of edits that you do. See *below*.
2. Hack away! See *below*
3. When finished, push your feature branch to your own Github repo, and *create a pull request*.

This way of working helps to keep work well organized and the history as clear as possible.

**Making a new feature branch**

First, fetch new commits from the *upstream* repository:

```bash
git fetch upstream
```

Then, create a new branch based on the master branch of the upstream repository:

```bash
git checkout -b my-new-feature upstream/master
```

**The editing workflow**

**Overview**

```
# hack hack
git status # Optional
git diff # Optional
git add modified_file
git commit
```

(continues on next page)
In more detail

1. Make some changes. When you feel that you’ve made a complete, working set of related changes, move on to
the next steps.

2. Optional: Check which files have changed with `git status`. You’ll see a listing like this one:

```bash
# On branch my-new-feature
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working directory)
#
# modified: README
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# INSTALL
# no changes added to commit (use "git add" and/or "git commit -a")
```

3. Optional: Compare the changes with the previous version using with `git diff`. This brings up a simple text
browser interface that highlights the difference between your files and the previous version.

4. Add any relevant modified or new files using `git add modified_file`. This puts the files into a staging
area, which is a queue of files that will be added to your next commit. Only add files that have related, complete
changes. Leave files with unfinished changes for later commits.

5. To commit the staged files into the local copy of your repo, do `git commit`. At this point, a text editor will
open up to allow you to write a commit message. After saving your message and closing the editor, your commit
will be saved. For trivial commits, a short commit message can be passed in through the command line using
the `-m` flag.

6. Push the changes to your forked repo on GitHub:

```bash
git push origin my-new-feature
```

**Note:** Assuming you have followed the instructions in these pages, git will create a default link to your GitHub repo
called `origin`. In git >= 1.7 you can ensure that the link to origin is permanently set by using the `--set-upstream`
option:

```bash
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in your own github
repo. Subsequent push calls are then simplified to the following:

```bash
git push
```

You have to use `--set-upstream` for each new branch that you create.

It may be the case that while you were working on your edits, new commits have been added to `upstream` that affect
your work. In this case, follow the `Rebasing on master` section of this document to apply those changes to your branch.

### 3.1. Contributing to Simphony
Asking for your changes to be merged with the main repo

When you feel your work is finished, you can create a pull request (PR).

If your changes involve modifications to the API or addition/modification of a function, you should be sure to emphasize this in the pull request. This may generate changes and feedback. It might be prudent to start with this step if your change may be controversial or make existing scripts not backward-compatible.

Rebasing on master

This updates your feature branch with changes from the upstream Simphony github repo. If you do not absolutely need to do this, try to avoid doing it, except perhaps when you are finished. The first step will be to update the remote repository with new commits from upstream:

```
git fetch upstream
```

Next, you need to update the feature branch:

```
# go to the feature branch
git checkout my-new-feature
# make a backup in case you mess up
git branch tmp my-new-feature
# rebase on upstream master branch
git rebase upstream/master
```

If you have made changes to files that have changed also upstream, this may generate merge conflicts that you need to resolve. See below for help in this case.

Finally, remove the backup branch upon a successful rebase:

```
git branch -D tmp
```

Note:  Rebasings on master is preferred over merging upstream back to your branch. Using `git merge` and `git pull` is discouraged when working on feature branches.

Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in Git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:
If you didn’t actually mess up but there are merge conflicts, you need to resolve those. This can be one of the trickier things to get right.

**Additional things you might want to do**

**Deleting a branch on GitHub**

```bash
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

(Note the colon : before test-branch. See also: https://github.com/guides/remove-a-remote-branch

**A Guide to Simphony Documentation**

When using Sphinx in combination with the numpy conventions, you should use the `sphinx.ext.napoleon` extension so that your docstrings will be handled correctly. For example, Sphinx will extract the Parameters section from your docstring and convert it into a field list. Using `sphinx.ext.napoleon` will also avoid the reStructuredText errors produced by plain Sphinx when it encounters numpy docstring conventions like section headers (e.g. `------------`) that sphinx does not expect to find in docstrings.

Note that for documentation within Simphony, it is not necessary to do `import simphony` at the beginning of an example. However, some sub-modules, such as `library`, are not imported by default, and you have to include them explicitly:

```python
import simphony.library.ebeam
```

after which you may use it:

```python
simphony.library.ebeam.ebeam_wg_integral_1550(...)
```
All inline documentation should adhere to the numpydoc formatting standard.

Sometimes there is a class attribute that you’d like documented but because it isn’t a function, it has no docstring. Luckily, sphinx allows us to autodocument attributes like so:

```python
#: Indicates some unknown error.
API_ERROR = 1
```

Using multiple `#:` lines before any assignment statement, or a single `#:` comment to the right of the statement, work effectively the same as docstrings on objects picked up by autodoc. This includes handling inline rST, and auto-generating an rST header for the variable name; there’s nothing extra you have to do to make that work (thanks, abarnert for the tip!).

Many of the library components are best documented with an accompanying picture for cross-referencing port names and location. You can add images to your inline documentation using rST syntax as long as the actual image resides in the docs/source directory. References can be expressed as absolute paths with respect to the source directory (see Stack Overflow for more details).

In keeping with the numpydoc standard, class initialization parameters should be documented in the class docstring, not under `__init__()`. Names of classes, objects, constants, etc. should typically be linked to the more detailed documentation of the referred object.

Examples presented are prefixed with the Python prompt `>>>`.

The docs are written in reST. There is a nice syntax guide with guidelines that we follow in the documentation [here](https://thomas-cokelaer.info/tutorials/sphinx/rest_syntax.html). The majority of the documentation is generated from python docstrings written using the NumPy documentation format.

**Building the Simphony API and reference docs**

We currently use Sphinx for generating the API and reference documentation for Simphony. .. You will need Sphinx >= 2.2.0.

If you only want to get the documentation, note that pre-built versions can be found at

https://simphonyphotonics.readthedocs.io/

in several different formats.

**Instructions**

Simphony has dependencies on other Python projects. Be sure to install its requirements, listed in `requirements.txt`, or simply install using pip:

```bash
pip install .
```

If using Python virtual environments (via e.g. conda, virtualenv or the `venv` module), installing Simphony into a new virtual environment is recommended. All of the necessary dependencies for building the Simphony docs can be installed with:

```bash
$ pip install .[docs]
```

Now you are ready to generate the docs, so write:
In the docs/ directory. If all goes well, this will generate a build/html subdirectory containing the built documentation.

To build the PDF documentation, do instead:

```
make latex
make -C build/latex all-pdf
```

You will need to have Latex installed for this, inclusive of support for Greek letters. For example, on Ubuntu xenial texlive-lang-greek and cm-super are needed. Also latexmk is needed on non-Windows systems.

The documentation for Simphony distributed at https://simphonyphotonics.readthedocs.io/ in html and pdf format is automatically built from the .readthedocs.yml configuration file. The documentation hosted at https://simphonyphotonics.readthedocs.io/ is automatically rebuilt any time a version is activated or a commit to master is made, see How to Prepare a Release.

**Sphinx extensions**

Simphony’s documentation uses several Sphinx extensions. While the code docstrings are written using the numpydoc standard, we actually use Sphinx’s built-in napolean extension to parse our files. Napolean has been included in the standard Sphinx since version 1.3, so no special parsing extensions are required to generate this documentation.

**Releasing a Version**

**For Developers**

This package is available on PyPI and updates are regularly pushed as “minor” or “micro” (patch) versions. Before submitting any pull requests, however, you should ensure that a pip installation of your updated package installs and functions properly. To test this, try installing your package locally by removing all installed versions of Simphony (by running `pip3 uninstall simphony` repeatedly until no installations remain) and running the following commands (from Simphony’s toplevel directory):

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

**For Maintainers**

Remember that all changes are to be integrated through pull requests. Development work should be done in branches or forks of the repository. Once implemented (and tested on their own), these pull requests should be merged into the “master” branch for full testing with the whole program. Each time the package is released on PyPI, the package should have a pull request opened to its corresponding release branch (release-MAJOR.MINOR.x). The hierarchy is then as follows:

- release-..x (stable branch)
- master (integration and final testing)
- feature-name (feature development and bug fixes)

Even if you are the lone developer, we follow the methodology [here](https://softwareengineering.stackexchange.com/a/294048).

3.1. Contributing to Simphony
Be sure to update the version number manually before pushing each new version to PyPI. Also be sure to amend the changelog. Versions can be pushed to PyPI using the commands:
```
\python3 setup.py sdist bdist_wheel python3 -m twine upload dist/*
```
Eventually, as the project grows, we will work up to using the methods detailed below, retained for future reference.

### How to Prepare a Release

This file gives an overview of what is necessary to build binary releases for Simphony.

#### Supported platforms and versions

**Platform Independent**

As a relatively straightforward Python-only package, Simphony doesn’t require any special builds for different operating systems and should be universally installable via pip.

**Tool chain**

We build all our wheels locally on Linux.

**Building source archives and wheels**

- Python(s) from [python.org](http://python.org) or linux distro.
- virtualenv (pip)

**Building docs**

Building the documents requires the items listed below and reproduced in a doc_requirements.txt file.

- Sphinx (pip)

**Uploading to PyPI**

- twine (pip)

**What is released**

**Wheels**

Simphony ought to be OS-independent. Hence, a `none-any` built wheel is included in the release.
Other

- Changelog

Source distribution

We build source releases in both .zip and .tar.gz formats.

Release process

Make sure current branch builds a package correctly

```
git clean -fxd
python setup.py bdist
python setup.py sdist
```

**Note:** The following steps are repeated for the beta(s), release candidates(s) and the final release.

Check deprecations

Before the release branch is made, it should be checked that all deprecated code that should be removed is actually removed, and all new deprecations say in the docstring or deprecation warning at what version the code will be removed.

Check the changelog

Check that the changelog, which is handwritten, is up-to-date.

After the first paragraph summary/overview, perhaps mention some of the following highlights:

- major new features
- deprecated and removed features
- supported Python versions
- outlook for the near future

A template for a typical changelog is as follows:

```markdown
## [MAJOR.MINOR.PATCH] - YEAR-MM-DD

This section provides an overview/summary of the release.

Any other highlights, as necessary.

### Added
- [#<val>](<link-to-issue>) Description of what was added.

### Changed
- List of changes.
```

(continues on next page)
Update the release status and create a release “tag”

Make sure all the code to be included in the release is up to date and pushed to master.

Go to https://github.com/BYUCamachoLab/simphony/releases, the main Simphony repository in GitHub, and draft a new release. The name of the release should be the version number, in the following format:

\[v<\text{MAJOR}>.<\text{MINOR}>.<\text{PATCH}> \quad \# \ e.g. \ v0.3.0\]

Update the version of the master branch

Increment the release number in `simphony/__init__.py`. Release candidates should have “rc1” (or “rc2”, “rcN”) appended to the X.Y.Z format.

Trigger the wheel builds

We use `setuptools` and `wheel` to package Simphony. Make sure you have the latest version installed:

```
python3 -m pip install --user --upgrade setuptools wheel
```

In the same directory as `setup.py`, run the following command:

```
python3 setup.py sdist --formats=gztar,zip bdist_wheel
```

It will create the `dist` directory and place within it the `*.zip` and `*.tar.gz` source releases, as well as the built distribution `*.whl`. Since Simphony is not OS-specific (at least for now), the single wheel should be good for any platform.

Build and archive documentation

Do:

```
cd doc/  
make dist
```

to check that the documentation is in a buildable state. Then, after tagging a release in GitHub, activate the documentation version online at the web interface at ReadTheDocs by using the git tag that is the release version.
Update PyPI

The wheels and source should be uploaded to PyPI.

You should upload the wheels first, and the source formats last, to make sure that pip users don’t accidentally get a source install when they were expecting a binary wheel.

```
$ git clean -fxd  # to be safe
$ python setup.py sdist --formats=gztar,zip  # to check
# python setup.py sdist --formats=gztar,zip upload --sign
```

This will ask for your key PGP passphrase, in order to sign the built source packages.

Upload files to github

Once the wheels have been built without errors, go to https://github.com/BYUCamachoLab/simphony/releases, the main Simphony repository in GitHub, and update the release by clicking Edit next to the appropriate release.

The subsequent page has has two locations to add files and content, using an editable text window and as file uploads.

- Cut and paste the docs/changelog/0.3.0-changelog.md file contents into the text window.
- Upload dist/simphony-0.3.0.tar.gz as a binary file.
- Upload dist/simphony-0.3.0.zip as a binary file.
- Upload the file docs/changelog/0.3.0-changelog.md.
- Hit the {Publish,Update} release button at the bottom.

Step-by-Step Directions

This file contains a walkthrough of the Simphony 0.3.0 release on Linux. The commands can be copied into the command line, but be sure to replace 0.3.0 by the correct version.

Release Walkthrough

Note that in the code snippets below, upstream refers to the root repository on github and origin to a fork in your personal account. You may need to make adjustments if you have not forked the repository but simply cloned it locally. You can also edit .git/config and add upstream if it isn’t already present.

Update Release documentation

The file docs/changelog/0.3.0-changelog.md should be updated to reflect the final list of changes. For now, this is a manual process. Below is a template of what sections to include:

```markdown
## [MAJOR.MINOR.PATCH] - YEAR-MM-D

This section provides an overview/summary of the release.

### Added
- [#<val>](<link-to-issue>) Description of what was added.

### Changed
```

(continues on next page)
### Prepare the release commit

Checkout the branch for the release, make sure it is up to date, and clean the repository:

```bash
$ git checkout master
$ git pull upstream master
$ git clean -xdfq
```

Sanity check:

```bash
$ pytest
```

Push this release directly onto the end of the master branch. This requires write permission to the Simphony repository:

```bash
$ git push upstream master
```

### Build source releases and wheels

**Note:** Simphony gets published automatically to PyPI when a new version is tagged in GitHub. The following is the process followed by the GitHub Actions workflow to publish to PyPI, and does NOT need to be performed manually.

We use `setuptools` and `wheel` to package Simphony. Make sure you have the latest version installed:

```bash
python3 -m pip install --user --upgrade setuptools wheel
```

In the same directory as `setup.py`, run the following command:

```bash
python3 setup.py sdist --formats=gztar,zip bdist_wheel
```

It will create the `dist` directory and place within it the `*.zip` and `*.tar.gz` source releases, as well as the built distribution `*.whl`. Since Simphony is not OS-specific (at least for now), the single wheel should be good for any platform.

### Tag the release

**Note:** You will need at least Write access on the main Simphony repository to create a GitHub release for Simphony.

Go to https://github.com/BYUCamachoLab/simphony/releases, the main Simphony repository in GitHub, and draft a new release. The name of the release should be the version number, which you should have verified is the version number defined throughout the package.

```bash
v0.3.0
```
Upload files to github

There are two locations to add files and content, using an editable text window and as file uploads.

- Cut and paste the docs/changelog/0.3.0-changelog.md file contents into the text window.
- Upload dist/simphony-0.3.0.tar.gz as a binary file.
- Upload dist/simphony-0.3.0.zip as a binary file.
- Upload the file docs/changelog/0.3.0-changelog.md.
- Hit the {Publish,Update} release button at the bottom.

Upload to PyPI

Upload to PyPI using twine. A recent version of twine of is needed.

```bash
$ cd ../simphony
$ python3 -m twine upload dist/*whl
$ python3 -m twine upload dist/numpy-1.14.5.zip  # Upload last.
```

If one of the commands breaks in the middle, which is not uncommon, you may need to selectively upload the remaining files because PyPI does not allow the same file to be uploaded twice. The source file should be uploaded last to avoid synchronization problems if pip users access the files while this is in process. Note that PyPI only allows a single source distribution, here we have chosen the zip archive.

Upload documents to simphonyphotonics.readthedocs.io

Documentation in the repository when the version is tagged (released) should already be up to date. If you have maintainer privileges on the Simphony ReadTheDocs page, you should add the new release to the Active Versions section using the git tag name.

Report a Bug

File bug reports or feature requests, and make contributions (e.g. code patches), by opening a “new issue” on GitHub:

- Simphony Issues: https://github.com/BYUCamachoLab/simphony/issues

Please give as much information as you can in the ticket. It is extremely useful if you can supply a small self-contained code snippet that reproduces the problem. Also specify the component or module and the version you are using.

Our workflow, based strongly on the NumPy project, is in development-workflow.
• View the API: API
• License agreement: License
• How we name stuff: Glossary

4.1 API

4.1.1 simphony

A Simulator for Photonic circuits

License

MIT License

Copyright (c) 2019- Simphony Project Contributors and others (see AUTHORS.txt)
The resources, libraries, and some source files under other terms (see NOTICE.txt).

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
4.1.2 simphony.connects

Code for s-parameter matrix cascading uses the scikit-rf implementation. Per their software license, the copyright notice is reproduced below:

Copyright (c) 2010, Alexander Arsenovic All rights reserved.

Copyright (c) 2017, scikit-rf Developers All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the scikit-rf nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

```
simphony.connect.connect_s(A, k, B, l)
```

connect two n-port networks’ s-matrices together. Specifically, connect port k on network A to port l on network B. The resultant network has nports = (A.rank + B.rank-2). This function operates on, and returns s-matrices.

The function `connect` operates on `Network` types.

- :param A: S-parameter matrix of A, shape is fxnxn :type A: numpy.ndarray
- :param k: port index on A (port indices start from 0) :type k: int
- :param B: S-parameter matrix of B, shape is fxnxn :type B: numpy.ndarray
- :param l: port index on B :type l: int

Returns C – new S-parameter matrix

Return type numpy.ndarray

Notes

internally, this function creates a larger composite network and calls the `innerconnect_s` function. see that function for more details about the implementation

See also:

- `connect` operates on `Network` types
- `innerconnect_s` function which implements the connection connection algorithm

```
simphony.connect.innerconnect_s(A, k, l)
```

connect two ports of a single n-port network’s s-matrix. Specifically, connect port k to port l on A. This results in a (n-2)-port network. This function operates on, and returns s-matrices. The function `innerconnect` operates on `Network` types.

- :param A: S-parameter matrix of A, shape is fxnxn :type A: numpy.ndarray
- :param k: port index on A (port indices start from 0) :type k: int
- :param l: port index on A :type l: int
Returns C – new S-parameter matrix

Return type numpy.ndarray

Notes

The algorithm used to calculate the resultant network is called a ‘sub-network growth’, can be found in\(^1\). The original paper describing the algorithm is given in\(^2\).

References

4.1.3 simphony.elements

This package contains the base classes for defining models.

```python
class simphony.elements.Model
    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 the attributes of an element (see Attributes). Following the general EAFP coding style of Python, errors will only be raised when an unimplemented function is called, not when the class instance is created.

    pins
        A tuple of all the default pin names of the device. Length of default tuple should be equal to the number of ports on the device.
        
        Type tuple of str

    freq_range
        A tuple of the valid frequency bounds for the element in the order (lower, upper). Can be made (-infty, infy) be setting to (None, None).
        
        Type tuple of float

    monte_carlo_s_parameters(freq)

    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 freq (np.ndarray) – 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

    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.

```python
s_parameters(freq)
```

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

**Parameters**

- `freq` *(np.ndarray)* – 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.

### 4.1.4 simphony.netlist

This package contains the base classes for defining circuits.

```python
class simphony.netlist.Element(model, name=None)
```

Represents an instantiation of some model in a circuit.

Unites a `Model` with a `PinList` to allow unique instances to be instantiated within a `Subcircuit`.

**Parameters**

- `model` *(simphony.elements.Model)* – The model this element represents.

- `name` *(str, optional)* – Unique string identifying the element, autogenerated if not specified.

**name**

The read-only name of the element, unique within each `Subcircuit`. If not specified on instantiation, it is autogenerated.

**Type** `str`

**model**

A reference to a `Model` instance (NOTe: it must be an instance, not a class reference).

**Type** `simphony.elements.Model`

**pins**

A `PinList`, generated automatically from the model, with pins renameable after instantiation.

**Type** `simphony.netlist.PinList`
Notes

Deep copying doesn’t have the full effect on this object. Since models are supposed to be universal throughout a simulation (thereby reducing cache and comparison time), when this object is deep copied, the model attribute remains as a reference to the same former model object.

class simphony.netlist.ElementList
Bases: object

Maintains an ordered dict. If an update to an existing key is attempted, the update fails. Keys must be deleted before being used.

Dictionary is a mapping of names (type: str) to elements or blocks (type: Element or Subcircuit).

Allows for access to blocks within the subcircuit by name or index, similar to a dictionary or list.

keys()
Returns the keys of the ElementList as a list of strings.

    Returns: keys – The keys (or, names of Element instances) of ElementList.

    Return type: list of str

class simphony.netlist.Netlist
Bases: object

Maintains a list of all connections, or “nets”, in a circuit.

nets
Nets is a list of connections, stored as a list of two Pins.

    Type: list of list

class simphony.netlist.Pin(pinlist, name)
Bases: object

A class representing a pin on a unique element instance.

Note that these are not the pins defined in Models, but are created from the names defined there.

Parameters

    • pinlist (simphony.elements.PinList) – The PinList this pin resides in.
    • name (str) – The name of the pin.

Notes

A Pin can only exist in one PinList at a time. Moving a Pin into another PinList will result in the automatic and silent change of the pinlist reference.

property element
    Returns the element to which this pin belongs by tracing the path to PinList, which ought to hold a reference to an Element.

class simphony.netlist.PinList(element, *pins)
Bases: object

A list of pins belonging to an Element, indexed the same way the s-parameters of a Model are indexed.

PinList maintains unique Pin names within its list. Pins can also be accessed by index instead of name.
• **element** (*simphony.elements.Element*) – The element this PinList defines the pins for.

• **pins** (*str or Pin*) – Number of unnamed arguments is not limited; each corresponds to a new Pin in the PinList. If str, Pin is created. If Pin, its pinlist attribute is updated to point to this PinList.

**element**

    The **Element** the PinList belongs to.

    **Type**  simphony.elements.Element

**pins**

    A list of **Pin** objects, indexed in the same order as the s-parameters of the **Model** it represents.

    **Type**  list of simphony.element.Pin

**Notes**

If renaming pins, the assigned value must be a string.

**Note:** If a PinList contains two Pins with the same string name and access by string value is attempted, a **LookupError** is raised complaining that the name is ambiguous.

**Warning:** Adding two PinLists together will change the pinlist reference of the pins they contain to point to the new result. This is because Pins can only be referenced by one PinList at a time. Inserting them into a new PinList automatically and silently changes their references.

**Examples**

```python
>>> pinlist = PinList(None, 'n1', 'n2', 'n3')
>>> pinlist.pins
[<'n1' simphony.netlist.Pin object from at 0x7f1098ef39b0>, <'n2' simphony.netlist.Pin object from at 0x7f1098ef3c88>, <'n3' simphony.netlist.Pin object from at 0x7f108aec6160>]
>>> pinlist.n2 = 'out1'
>>> pinlist.pins
[<'n1' simphony.netlist.Pin object from at 0x7f1098ef39b0>, <'out1' simphony.netlist.Pin object from at 0x7f1098ef3c88>, <'n3' simphony.netlist.Pin object from at 0x7f108aec6160>]
>>> pinlist.pins = ('out', 'in', 'mix')
>>> pinlist.pins = ('n1')

**append** (*pin*)

    Takes a pin argument (string or Pin) and creates a Pin object.

    **Parameters**  **pin** (*str or Pin*) – The pin to be normalized to a Pin.

**contains** (*pin*)

    **Parameters**  **pin** (*str or Pin*) – The pin to verify is in the list.

    **Returns**  True if the pin is in the list.

    **Return type**  bool
**index** *(pin)*
Given a Pin object, returns its index or position in the PinList.

- **Parameters**
  - *pin* *(simphony.netlist.Pin)* – The pin object to be found in the PinList.
- **Returns**
  - *idx* – The index of the pin passed in.
- **Return type**
  - int

**property pinnames**
Get the names of the pins in the PinList, in order.

- **Returns**
  - *names* – The formal names of each pin in the pinlist.
- **Return type**
  - tuple of str

**pop** *(idx=-1)*
Removes a pin from the pinlist by index (or, the last inserted pin by default).

- **Parameters**
  - *idx* *(int, optional)* – The index of the pin to remove from the list. If none, removes the last item in the list.

**remove** *(pins)*
Removes a pin from the pinlist by name or value.

- **Parameters**
  - *pins* *(str or Pin)* – Variable length argument list; the pins to be removed from the circuit.

**class** *simphony.netlist.Subcircuit*(name=None)*
Bases: object

This implements a subcircuit that can be constructed and reused throughout the circuit.

- **Parameters**
  - *name* *(str)* – A name for identifying the subcircuit.

**name**
A formal name for the Subcircuit *(None allowed).*

- **Type**
  - str

**elements**

- **Type**
  - list of elements

**connections**

- **Type**
  - netlist

**pins**

- **Type**
  - the new pins to use as the pins of the subcircuit

**nets**

**add** *(elements)*
Adds elements to a subcircuit.

- **Parameters**
  - *elements* *(list of tuples)* – A list of elements to be added. Tuples are of the form *(name, block)*, where *name* is a unique string identifying the element in the subcircuit and *block* can be an instance of some element *(i.e. a subclass of simphony.elements.Element)* or another subcircuit.
- **Returns**
  - *added* – A list of object references to elements added to the subcircuit. Insertion order is preserved *(order of the list is the same as the order elements were added).*
- **Return type**
  - list
Raises **TypeError** – If `blocks` is not a list.

`connect(element1, pin1, element2, pin2)`

Connect two elements with a net.

Netlists are unique to and stored by a Subcircuit object. This means net identifiers (numbers, by default) can be reused between separate subcircuits but must be unique within each.

**Parameters**

- `element1` –
- `node1` –
- `element2` –
- `node2` –

`connect_many(conns)`

A convenience function for connecting many nets at once.

**Parameters**

`conns` *(list of tuple)* – A list of tuples, each formed as a tuple of arguments in the same order as that accepted by `connect`.

`to_spice()`

Perhaps this shouldn’t be built into Subcircuit, maybe an adapter class or some translator instantiated with a Subcircuit that iterates through and creates a netlist.

**property** `wl_bounds`

Returns a tuple of the valid wavelength range.

### 4.1.5 simphony.persist

This package contains handy functions for exporting models created by other libraries to a format that can be used by any simphony installation, regardless of whether the creating library is installed locally.

Static models can be exported; that is to say, models that implement dynamic functions like `monte_carlo_s_parameters()` will have those functions ignored, as they are highly model-dependent. Static information, however, such as the scattering parameters over a range of wavelengths, can be exported easily and models recreated when imported. Other model attributes, such as pin names and the valid frequency range, are also exported.

**Warning:** Models are pickled files with a ‘.mdl’ extension. Note that pickle has an inherent security risk, so if you do not trust the source of the data, do not load the file!

`simphony.persist.export_model(model, filename, wl=None, freq=None)`

Exports a simphony model (using pickle) for the given frequency/wavelength range to a ‘.mdl’ file.

Must include either the wavelength or frequency argument. If both are included, defaults to frequency argument.

**Parameters**

- `model` *(Model)* – Any class inheriting from simphony.elements.Model
- `filename` *(str)* – The filename (may include path to directory) to save the model to. Note that the suffix ‘.mdl’ will be appended to the filename.
- `wl` *(ndarray, optional)* – Wavelengths you want to save sparameters for (in meters).
- `freq` *(ndarray, optional)* – Frequencies you want to save sparameters for (in Hz).
Examples

We can write a model for a `ebeam_wg_integral_1550` instantiated with a length of 100 nanometers to a file named `wg100nm.mdl`.

```python
>>> import numpy as np
>>> from simphony.library.ebeam import ebeam_wg_integral_1550
>>> wg1 = ebeam_wg_integral_1550(100e-9)
>>> export_model(wg1, 'wg100nm', wl=np.linspace(1520e-9, 1580e-9, 51))
```

`simphony.persist.import_model(filename, force=False)`

Imports a model from file

Parameters

- **filename** (`str`) – The filename (may include path to directory) to load the model from.

Returns

- **model** – A class that inherits from `simphony.elements.Model` that is the reconstructed model.

Return type

- **class**

Examples

```python
>>> waveguide_100nm = import_model('wg100nano.mdl')
>>> wg = waveguide_100nm()
>>> s = wg.s_parameters(np.linspace(wl2freq(1540e-9), wl2freq(1560e-9), 51))
```

### 4.1.6 simphony.simulation

This package contains the base classes for running simulations.

```python
class simphony.simulation.MonteCarloSimulationResult (freq, smat, runs)
Bases: simphony.simulation.SimulationResult

Parameters

- **freq** (`np.ndarray`) –
- **smat** (`simphony.simulation.ScatteringMatrix`) –
- **runs** (`int`) –

```python
data (inp, outp, run, dB=False)

Parameters

- **inp** (`str or Pin`) – Input pin.
- **outp** (`str or Pin`) – Output pin.

class simphony.simulation.MonteCarloSweepSimulation (circuit: simphony.netlist.Subcircuit, start: float = 1.5e-06, stop: float = 1.6e-06, num: int = 2000, mode='wl')

Bases: simphony.simulation.SweepSimulation

A monte carlo sweep simulation.

Parameters
• **circuit** (*Subcircuit*) – The circuit to be simulated.

• **start** (*float*) – The start wavelength (in meters) or frequency (in Hz).

• **stop** (*float*) – The stop wavelength (in meters) or frequency (in Hz).

• **num** (*int*) – The number of sampled points.

• **mode** (*str*) – Defines sweep range mode; either ‘wl’ for wavelength (m) or ‘freq’ for frequency (Hz).

**static connect_circuit** (*netlist*)

Connects the s-matrices of a photonic circuit given its Netlist and returns a single ‘SimulatedComponent’ object containing the frequency array, the assembled s-matrix, and a list of the external nets (negative integers).

**Parameters**

• **component_list** (*List[SimulatedComponent]*) – A list of the components to be connected.

• **net_count** (*int*) – The total number of internal nets in the component list.

**Returns**

combined – After the circuit has been fully connected, the result is a single ComponentSimulation with fields f (frequency), s (s-matrix), and nets (external ports: negative numbers, as strings).

**Return type** ScatteringMatrix

**Notes**

This function doesn’t actually store combined on each iteration through the netlist. That’s because the Pin objects can only reference one PinList at a time, which in turn can only reference one Element. Since we transferring the actual Pin objects between lists, keeping a reference to the Pin also keeps a reference to the combined Element alive. Hence, we track pins but not the SimulationResult.

**simulate** (*runs=10*)

**Parameters**

• **runs** (*int, optional*) – The number of monte carlo iterations to run (default 10).

**static validate_models** (*models, freq*)

Ensures all models are valid over the specified frequency range.

**Parameters**

• **models** (*list*) – A list of the model objects to be verified.

• **freq** (*np.ndarray*) – The array of frequency values the simulation is defined over.

**Raises**

• **NotImplementedError** – If a model does not have a class attribute freq_range defining the valid frequency range for the model.

• **ValueError** – If the simulation frequencies are outside of the range of the valid frequencies for a model.

**class** simphony.simulation.MultiInputSweepSimulation (*circuit: simphony.netlist.Subcircuit, start: float = 1.5e-06, stop: float = 1.6e-06, num: int = 2000, mode='wl')

**Bases:** simphony.simulation.SweepSimulation
**static connect_circuit (netlist)**

Connects the s-matrices of a photonic circuit given its Netlist and returns a single ‘SimulatedComponent’ object containing the frequency array, the assembled s-matrix, and a list of the external nets (negative integers).

**Parameters**

- **component_list (List[SimulatedComponent])** – A list of the components to be connected.
- **net_count (int)** – The total number of internal nets in the component list.

**Returns combined** – After the circuit has been fully connected, the result is a single ComponentSimulation with fields f (frequency), s (s-matrix), and nets (external ports: negative numbers, as strings).

**Return type** ScatteringMatrix

**Notes**

This function doesn’t actually store combined on each iteration through the netlist. That’s because the Pin objects can only reference one PinList at a time, which in turn can only reference one Element. Since we transferring the actual Pin objects between lists, keeping a reference to the Pin also keeps a reference to the combined Element alive. Hence, we track pins but not the SimulationResult.

**simulate ()**

Runs the simulation on the object’s circuit.

**Returns sim** – A loaded SweepSimulationResult object.

**Return type** SweepSimulationResult

**static validate_models (models, freq)**

Ensures all models are valid over the specified frequency range.

**Parameters**

- **models (list)** – A list of the model objects to be verified.
- **freq (np.ndarray)** – The array of frequency values the simulation is defined over.

**Raises**

- **NotImplementedError** – If a model does not have a class attribute freq_range defining the valid frequency range for the model.
- **ValueError** – If the simulation frequencies are outside of the range of the valid frequencies for a model.

**class symphony.simulation.Simulation (circuit: symphony.netlist.Subcircuit)**

**Bases: object**

Once a simulation is run, it is completely decoupled from the circuit which created it. Its pins, while bearing the same name, are unique objects.

**circuit**

A simulation is instantiated with a completed circuit.

**Type** symphony.netlist.Subcircuit

**class symphony.simulation.SimulationResult (pinlist=None)**

**Bases: object**

A simulated block of a circuit; can represent either elements or entire subcircuits.
It is used by Simulation in order to store s-parameters of recursively included subcircuits and elements while cascading all blocks into one final component representing the circuit as a whole.

**Parameters**

- **component** *(Component, optional)* – A component to initialize the data members of the object.

- **pins**

  An ordered tuple of the external pin names of the simulated component.

  **Type** `simphony.netlist.PinList`

**class** `simphony.simulation.SinglePortSweepSimulation`

**Bases:** `simphony.simulation.SweepSimulation`

**static connect_circuit** *(netlist)*

Connects the s-matrices of a photonic circuit given its Netlist and returns a single ‘SimulatedComponent’ object containing the frequency array, the assembled s-matrix, and a list of the external nets (negative integers).

**Parameters**

- **component_list** *(List[SimulatedComponent])* – A list of the components to be connected.

- **net_count** *(int)* – The total number of internal nets in the component list.

**Returns**

- **combined** – After the circuit has been fully connected, the result is a single ComponentSimulation with fields `f` (frequency), `s` (s-matrix), and nets (external ports: negative numbers, as strings).

  **Return type** ScatteringMatrix

**Notes**

This function doesn’t actually store combined on each iteration through the netlist. That’s because the Pin objects can only reference one PinList at a time, which in turn can only reference one Element. Since we transferring the actual Pin objects between lists, keeping a reference to the Pin also keeps a reference to the combined Element alive. Hence, we track pins but not the SimulationResult.

**simulate** ()

Runs the simulation on the object’s circuit.

  **Returns** `sim` – A loaded SweepSimulationResult object.

  **Return type** `SweepSimulationResult`

**static validate_models** *(models, freq)*

Ensures all models are valid over the specified frequency range.

**Parameters**

- **models** *(list)* – A list of the model objects to be verified.

- **freq** *(np.ndarray)* – The array of frequency values the simulation is defined over.

**Raises**

- **NotImplementedError** – If a model does not have a class attribute `freq_range` defining the valid frequency range for the model.

- **ValueError** – If the simulation frequencies are outside of the range of the valid frequencies for a model.
class simphony.simulation.SweepSimulation(
circuit: simphony.netlist.Subcircuit, start: float = 1.5e-06, stop: float = 1.6e-06, num: int = 2000, mode='wl')

Bases: simphony.simulation.Simulation

A swept simulation.

Parameters

• circuit (Subcircuit) – The circuit to be simulated.
• start (float) – The start wavelength (in meters) or frequency (in Hz).
• stop (float) – The stop wavelength (in meters) or frequency (in Hz).
• num (int, optional) – The number of sampled points.
• mode (str, optional) – Defines sweep range mode; either ‘wl’ for wavelength (m) or ‘freq’ for frequency (Hz).

freq

The frequency array over which the simulation is performed.

Type np.ndarray

static connect_circuit (netlist)

Connects the s-matrices of a photonic circuit given its Netlist and returns a single ‘SimulatedComponent’ object containing the frequency array, the assembled s-matrix, and a list of the external nets (negative integers).

Parameters

• component_list (List[SimulatedComponent]) – A list of the components to be connected.
• net_count (int) – The total number of internal nets in the component list.

Returns combined – After the circuit has been fully connected, the result is a single ComponentSimulation with fields f (frequency), s (s-matrix), and nets (external ports: negative numbers, as strings).

Return type ScatteringMatrix

Notes

This function doesn’t actually store combined on each iteration through the netlist. That’s because the Pin objects can only reference one PinList at a time, which in turn can only reference one Element. Since we transferring the actual Pin objects between lists, keeping a reference to the Pin also keeps a reference to the combined Element alive. Hence, we track pins but not the SimulationResult.

simulate ()

Runs the simulation on the object’s circuit.

Returns sim – A loaded SweepSimulationResult object.

Return type SweepSimulationResult

static validate_models (models,freq)

Ensures all models are valid over the specified frequency range.

Parameters

• models (list) – A list of the model objects to be verified.
• **freq** (*np.ndarray*) – The array of frequency values the simulation is defined over.

**Raises**

- **NotImplementedError** – If a model does not have a class attribute *freq_range* defining the valid frequency range for the model.
- **ValueError** – If the simulation frequencies are outside of the range of the valid frequencies for a model.

```python
class simphony.simulation.SweepSimulationResult(freq, smat)
Bases: simphony.simulation.SimulationResult
```

A simulation result for a swept simulation.

**Parameters**

- **freq** (*np.array*) – A numpy array of the frequency values in its simulation.
- **smat** (*ScatteringMatrix*) – A numpy array of the s-parameter matrix for the given frequency range.

```python
data(inp, outp, dB=False)
```

**Parameters**

- **inp** (*str or Pin*) – Input pin.
- **outp** (*str or Pin*) – Output pin.

### 4.1.7 simphony.tools

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

```python
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**

```python
simphony.tools.get_subclasses(cls)
```

Recursively gets all subclasses for a given class, even the subclasses of subclasses.

If a subclass resides in a model not imported by default by Simphony, those classes will not be returned. Libraries must be imported first for this function to be able to find those classes.

**Parameters** **cls** (*class*) – The class to find all the subclasses of.

**Yields** **subclass** (*class*) – Yields the next subclass from the generator.
**Notes**

To get a list of subclasses, simply use the following syntax:

```python
list(get_subclasses(klass))
```

**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**

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

Values without suffixes get converted to floats normally.

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

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

```python
>>> 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:

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

```python
>>> 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

4.2 License

MIT License

Copyright (c) 2019- Simphony Project Contributors and others (see AUTHORS.txt)

The resources, libraries, and some source files under other terms (see NOTICE.txt).

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

4.3 Glossary

compound structure Any structure that can be broken down into smaller, simpler parts. A subcircuit is an example of a compound structure; it contains simpler elements (or other compound structures) connected internally to form the overall larger structure.

decorator An operator that transforms a function. For example, a log decorator may be defined to print debugging information upon function execution:

```python
>>> def log(f):
...     def new_logging_func(*args, **kwargs):
...         print("Logging call with parameters:", args, kwargs)
...         return f(*args, **kwargs)
...     return new_logging_func
... return new_logging_func
```

Now, when we define a function, we can “decorate” it using log:

```python
>>> @log
... def add(a, b):
...     return a + b
```

Calling add then yields:
>>> add(1, 2)
Logging call with parameters: (1, 2) {}
3

dictionary  Resembling a language dictionary, which provides a mapping between words and descriptions thereof, a Python dictionary is a mapping between two objects:

```python
>>> x = {1: 'one', 'two': [1, 2]}
```

Here, `x` is a dictionary mapping keys to values, in this case the integer 1 to the string “one”, and the string “two” to the list `[1, 2]`. The values may be accessed using their corresponding keys:

```python
>>> x[1]
'one'
>>> x['two']
[1, 2]
```

Note that dictionaries are not stored in any specific order. Also, most mutable (see immutable below) objects, such as lists, may not be used as keys.

For more information on dictionaries, read the Python tutorial.

externals  Pins not connected internally within a circuit, thus available for input/output.

instance  A class definition gives the blueprint for constructing an object:

```python
>>> class House:
...
    wall_colour = 'white'
```

Yet, we have to build a house before it exists:

```python
>>> h = House()  # build a house
```

Now, `h` is called a House instance. An instance is therefore a specific realisation of a class.

iterable  A sequence that allows “walking” (iterating) over items, typically using a loop such as:

```python
>>> x = [1, 2, 3]
>>> [item**2 for item in x]
[1, 4, 9]
```

It is often used in combination with enumerate:

```python
>>> keys = ['a', 'b', 'c']
>>> for n, k in enumerate(keys):
...    print("Key %d: %s" % (n, k))
...
Key 0: a
Key 1: b
Key 2: c
```

list  A Python container that can hold any number of objects or items. The items do not have to be of the same type, and can even be lists themselves:

```python
>>> x = [2, 2.0, "two", [2, 2.0]]
```

The list `x` contains 4 items, each which can be accessed individually:
It is also possible to select more than one item at a time, using slicing:

```python
>>> x[0:2]  # or, equivalently, x[:2]
[2, 2.0]
```

In code, arrays are often conveniently expressed as nested lists:

```python
>>> np.array(([1, 2], [3, 4]))
array([[1, 2],
       [3, 4]])
```

For more information, read the section on lists in the Python tutorial. For a mapping type (key-value), see dictionary.

**method** A function associated with an object. For example, each ndarray has a method called `repeat`:

```python
>>> x = np.array([1, 2, 3])
>>> x.repeat(2)
array([1, 1, 2, 2, 3, 3])
```

**reference** If `a` is a reference to `b`, then `(a is b) == True`. Therefore, `a` and `b` are different names for the same Python object.

**self** Often seen in method signatures, `self` refers to the instance of the associated class. For example:

```python
>>> class Paintbrush:
...     color = 'blue'
...     
...     def paint(self):
...         print("Painting the city %s!" % self.color)
...     
... p = Paintbrush()
>>> p.color = 'red'
>>> p.paint()  # self refers to 'p'
Painting the city red!
```

**scattering matrix** See scattering parameters.

**scattering parameters** Scattering parameters describe the phase and amplitude relationship between the ports of an abstracted device or circuit. It is represented by a $N \times N$ matrix where $N$ is the number of inputs/outputs/ports of the device.

**s-matrix** See scattering parameters.

**s-parameters** See scattering parameters.

**tuple** A sequence that may contain a variable number of types of any kind. A tuple is immutable, i.e., once constructed it cannot be changed. Similar to a list, it can be indexed and sliced:

```python
>>> x = (1, 'one', [1, 2])
>>> x
(1, 'one', [1, 2])
```
A useful concept is “tuple unpacking”, which allows variables to be assigned to the contents of a tuple:

```python
>>> x, y = (1, 2)
>>> x, y = 1, 2
```

This is often used when a function returns multiple values:

```python
>>> def return_many():
...    return 1, 'alpha', None

>>> a, b, c = return_many()
>>> a, b, c
(1, 'alpha', None)
```

```python
>>> a
1
>>> b
'alpha'
```
S

simphony, 73
simphony.connect, 73
simphony.elements, 75
simphony.library.ebeam, 37
simphony.netlist, 76
simphony.persist, 80
simphony.simulation, 81
simphony.tools, 86
INDEX

A
add() (simphony.netlist.Subcircuit method), 79
append() (simphony.netlist.PinList method), 78

C
circuit (simphony.simulation.Simulation attribute), 83
connect() (simphony.netlist.Subcircuit method), 80
connect_circuit() (simphony.simulation.MonteCarloSweepSimulation static method), 82
connect_circuit() (simphony.simulation.MultiInputSweepSimulation static method), 82
connect_circuit() (simphony.simulation.SinglePortSweepSimulation static method), 84
connect_circuit() (simphony.simulation.SweepSimulation static method), 85
connect_many() (simphony.netlist.Subcircuit method), 80
connect_s() (in module simphony.connect), 74
connections (simphony.netlist.Subcircuit attribute), 79
contains() (simphony.netlist.PinList method), 78

ebeam_bdc_te1550 (class in simphony.library.ebeam), 37
ebeam_ddc_te1550 (class in simphony.library.ebeam), 39
ebeam_ddc_te1550 (class in simphony.library.siepic), 47
ebeam_gc_te1550 (class in simphony.library.ebeam), 40
ebeam_gc_te1550 (class in simphony.library.siepic), 48
ebeam_terminator_te1550 (class in simphony.library.ebeam), 42
ebeam_terminator_te1550 (class in simphony.library.siepic), 47
ebeam_wg_integral_1550 (class in simphony.library.ebeam), 43
ebeam_wg_integral_1550 (class in simphony.library.siepic), 47
ebeam_y_1550 (class in simphony.library.ebeam), 44
ebeam_y_1550 (class in simphony.library.siepic), 49
Element (class in simphony.netlist), 76
element (simphony.netlist.PinList attribute), 78
element() (simphony.netlist.Pin property), 77
ElementList (class in simphony.netlist), 77
elements (simphony.netlist.Subcircuit attribute), 79
export_model() (in module simphony.persist), 80

D
data() (simphony.simulation.MonteCarloSimulationResult method), 81
data() (simphony.simulation.SweepSimulationResult method), 86

ebeam_bdc_te1550 (class in simphony.library.ebeam), 37
ebeam_bdc_te1550 (class in simphony.library.siepic), 46
ebeam_dc_halfring_straight (class in simphony.library.siepic), 46
ebeam_dc_halfring_te1550 (class in simphony.library.ebeam), 38

F
circuit (simphony.simulation.Simulation attribute), 83
circuit (simphony.netlist.Subcircuit method), 80
circuit (simphony.netlist.PinList method), 78
connect() (simphony.netlist.Subcircuit method), 80
connect() (simphony.netlist.Pin property), 77
connect_many() (simphony.netlist.Subcircuit method), 80
connect_s() (in module simphony.connect), 74
connections (simphony.netlist.Subcircuit attribute), 79
contains() (simphony.netlist.PinList method), 78

ebeam_bdc_te1550 (class in simphony.library.ebeam), 37
ebeam_ddc_te1550 (class in simphony.library.ebeam), 39
ebeam_ddc_te1550 (class in simphony.library.siepic), 47
ebeam_gc_te1550 (class in simphony.library.ebeam), 40
ebeam_gc_te1550 (class in simphony.library.siepic), 48
ebeam_terminator_te1550 (class in simphony.library.ebeam), 42
ebeam_terminator_te1550 (class in simphony.library.siepic), 47
ebeam_wg_integral_1550 (class in simphony.library.ebeam), 43
ebeam_wg_integral_1550 (class in simphony.library.siepic), 47
ebeam_y_1550 (class in simphony.library.ebeam), 44
ebeam_y_1550 (class in simphony.library.siepic), 49
Element (class in simphony.netlist), 76
element (simphony.netlist.PinList attribute), 78
element() (simphony.netlist.Pin property), 77
ElementList (class in simphony.netlist), 77
elements (simphony.netlist.Subcircuit attribute), 79
export_model() (in module simphony.persist), 80

freq (simphony.simulation.SweepSimulation attribute), 85
data() (simphony.simulation.MonteCarloSimulationResult method), 81
data() (simphony.simulation.SweepSimulationResult method), 86

ebeam_bdc_te1550 (class in simphony.library.ebeam), 37
ebeam_ddc_te1550 (class in simphony.library.ebeam), 39
ebeam_ddc_te1550 (class in simphony.library.siepic), 46
ebeam_terminator_te1550 (class in simphony.library.ebeam), 41

ebeam_bdc_te1550 (class in simphony.library.ebeam), 37
ebeam_ddc_te1550 (class in simphony.library.ebeam), 39
ebeam_ddc_te1550 (class in simphony.library.siepic), 46
ebeam_terminator_te1550 (class in simphony.library.ebeam), 41

ebeam_wg_integral_1550 (class in simphony.library.ebeam), 43
ebeam_wg_integral_1550 (class in simphony.library.siepic), 47

ebeam_y_1550 (class in simphony.library.ebeam), 44
ebeam_y_1550 (class in simphony.library.siepic), 49

Element (class in simphony.netlist), 76
element (simphony.netlist.PinList attribute), 78
element() (simphony.netlist.Pin property), 77
ElementList (class in simphony.netlist), 77
elements (simphony.netlist.Subcircuit attribute), 79
export_model() (in module simphony.persist), 80

freq2wl() (in module simphony.tools), 86
def_range (simphony.elements.Model attribute), 75
freq_range (simphony.library.ebeam.ebeam_bdc_te1550 attribute), 37
def_range (simphony.library.ebeam.ebeam_bdc_te1550 attribute), 38
def_range (simphony.library.ebeam.ebeam_dc_halfring_te1550 attribute), 39
def_range (simphony.library.ebeam.ebeam_dc_halfring_te1550 attribute), 41
def_range (simphony.library.ebeam.ebeam_ebeam_terminator_te1550 attribute), 42
def_range (simphony.library.ebeam.ebeam_wg_integral_1550 attribute), 44
freq_range (symphony.library.ebeam.ebeam_y_1550 attribute), 45

G
get_subclasses() (in module symphony.tools), 86

I
import_model() (in module symphony.persist), 81
index() (symphony.netlist.PinList method), 78
innerconnect_s() (in module symphony.connect), 74
interpolate() (in module symphony.tools), 87

K
keys() (symphony.netlist.ElementList method), 77

M
Model (class in symphony.elements), 75
model (symphony.netlist.Element attribute), 76
module
  symphony, 73
  symphony.connect, 73
  symphony.elements, 75
  symphony.library.ebeam, 37
  symphony.netlist, 76
  symphony.persist, 80
  symphony.simulation, 81
  symphony.tools, 86
monte_carlo_s_parameters() (symphony.elements.Model method), 75
monte_carlo_s_parameters() (symphony.library.ebeam.ebeam_bdc_te1550 method), 37
monte_carlo_s_parameters() (symphony.library.ebeam.ebeam_dc_halfring_te1550 method), 38
monte_carlo_s_parameters() (symphony.library.ebeam.ebeam_gc_te1550 method), 39
monte_carlo_s_parameters() (symphony.library.ebeam.ebeam_y_1550 method), 40
monte_carlo_s_parameters() (symphony.library.ebeam.ebeam_y_1550 attribute), 41
monte_carlo_s_parameters() (symphony.library.ebeam.terminator_te1550 method), 42
monte_carlo_s_parameters() (symphony.library.ebeam.wg_integral_1550 method), 44
MonteCarloSimulationResult (class in symphony.simulation), 81
MonteCarloSweepSimulation (class in symphony.simulation), 81
MultiInputSweepSimulation (class in symphony.simulation), 82

N
name (symphony.netlist.Element attribute), 76
name (symphony.netlist.Subcircuit attribute), 79
Netlist (class in symphony.netlist), 77
nets (symphony.netlist.Netlist attribute), 77
nets (symphony.netlist.Subcircuit attribute), 79

P
Pin (class in symphony.netlist), 77
PinList (class in symphony.netlist), 77
pinnames() (symphony.netlist.PinList property), 79
pins (symphony.elements.Model attribute), 75
pins (symphony.library.ebeam.ebeam_bdc_te1550 attribute), 38
pins (symphony.library.ebeam.ebeam_dc_halfring_te1550 attribute), 39
pins (symphony.library.ebeam.ebeam_gc_te1550 attribute), 41
pins (symphony.library.ebeam.ebeam_terminator_te1550 attribute), 42
pins (symphony.library.ebeam.ebeam_wg_integral_1550 attribute), 44
pins (symphony.library.ebeam.ebeam_y_1550 attribute), 45
pins (symphony.library.iek.ebeam_bdc_te1550 attribute), 46
pins (symphony.library.iek.ebeam_dc_halfring_straight attribute), 47
pins (symphony.library.iek.ebeam_dc_te1550 attribute), 47
pins (symphony.library.iek.ebeam_terminator_te1550 attribute), 48
pins (symphony.library.iek.ebeam_wg_integral_1550 attribute), 49
pins (symphony.library.iek.ebeam_y_1550 attribute), 49
pins (symphony.netlist.Element attribute), 76
pins (symphony.netlist.PinList attribute), 78
pins (symphony.netlist.Subcircuit attribute), 79
pins (symphony.simulation.SimulationResult attribute), 84
pop() (symphony.netlist.PinList method), 79

R
regenerate_monte_carlo_parameters() (symphony.elements.Model method), 75
Index 97

regenerate_monte_carlo_parameters() module, 37
(simphony.library.ebeam.ebeam_bdc_te1550 method), 38
regenerate_monte_carlo_parameters() simphony.netlist module, 76
(simphony.library.ebeam.ebeam_dc_halfring_te1550 method), 39
regenerate_monte_carlo_parameters() simphony.persist module, 80
(simphony.library.ebeam.ebeam_dc_te1550 method), 40
regenerate_monte_carlo_parameters() simphony.simulation module, 81
(simphony.library.ebeam.ebeam_gc_te1550 method), 41
regenerate_monte_carlo_parameters() simphony.tools module, 86
(simphony.library.ebeam.ebeam_terminator_te1550 method), 42
regenerate_monte_carlo_parameters() simulate() (simphony.simulation.MonteCarloSweepSimulation method), 82
(simphony.library.ebeam.ebeam_wg_integral_1550 method), 44
regenerate_monte_carlo_parameters() simulate() (simphony.simulation.MultiInputSweepSimulation method), 83
(simphony.library.ebeam.ebeam_y_1550 method), 45
reduce() (simphony.netlist.PinList method), 79
s_parameters() (simphony.elements.Model method), 76
s_parameters() (simphony.library.ebeam.ebeam_bdc_te1550 method), 38
s_parameters() (simphony.library.ebeam.ebeam_dc_halfring_te1550 method), 39
s_parameters() (simphony.library.ebeam.ebeam_dc_te1550 method), 40
s_parameters() (simphony.library.ebeam.ebeam_gc_te1550 method), 42
s_parameters() (simphony.library.ebeam.ebeam_terminator_te1550 method), 43
s_parameters() (simphony.library.ebeam.ebeam_wg_integral_1550 method), 44
s_parameters() (simphony.library.ebeam.ebeam_y_1550 method), 45
simulation module, 73
simulate() (simphony.simulation.MonteCarloSweepSimulation method), 82
simulate() (simphony.simulation.MultiInputSweepSimulation method), 83
simulation (class in simphony.simulation), 83
SimulationResult (class in simphony.simulation), 83
SinglePortSweepSimulation (class in simphony.simulation), 84
SweepSimulation (class in simphony.simulation), 84
SweepSimulationResult (class in simphony.simulation), 86
to_spice() (simphony.netlist.Subcircuit method), 80
t0_validate_models() (simphony.simulation.MonteCarloSweepSimulation static method), 82
t0_validate_models() (simphony.simulation.MultiInputSweepSimulation static method), 83
t0_validate_models() (simphony.simulation.SinglePortSweepSimulation static method), 84
t0_validate_models() (simphony.simulation.SweepSimulation static method), 85
wl2freq() (in module simphony.tools), 87
wl_bounds() (simphony.netlist.Subcircuit property), 80

W

wl2freq() (in module simphony.tools), 87
wl_bounds() (simphony.netlist.Subcircuit property), 80

Index 97