SonPy Documentation

Release 1.1.0

May 19, 2022

CONTENTS:

CHAPTER

ONE

ABOUT SONPY

1.1 Credits

SonPy was developed in the Engineering Quantum Systems Group at Massachusetts Insitute of Technology in the spring/summer of 2018. Its first version was written by Niels Jakob Søe Loft (nsl@phys.au.dk) with great help from Bharath Kannan. The project was inspired by work by Daniel Becerra-Pérez and José E. Rayas-Sánchez, who were very kind to share their code with us. Further development was done by Megan Yamoah.

This documentation was generated by Sphinx. Remember to document your contributions to the code, and compile a new documentation (like this one) using Sphinx and the command `make latexpdf` from the `docs` directory. The new pdf documentation is found in the `_build` directory.

Version 1.0: Niels Jakob Søe Loft

Version 1.1: Megan Yamoah

1.2 Purpose

SonPy was developed in an effort to automate some of the tedious and time consuming work that is circuit design by creating a Python interface for the commercial simulation software Sonnet. It is designed to fit the specific needs of the EQuS Group. Therefore it covers only a subset of the functionality in Sonnet, but it is straightforward to include more of Sonnet's many possibilities. SonPy is written for Python 3.

Sonnet comes for both Linux and Windows, but SonPy is developed and tested only for the Windows version. However, it should be relatively straightforward to change the file paths and subprocess handling in SonPy to fit Linux standards.

1.3 License

CHAPTER

TWO

USER'S GUIDE

2.1 Introduction

SonPy is a Python package that allows the user to define, manipulate and simulate Sonnet projects. Sonnet is an electromagnetic circuit simluator from Sonnet Software, Inc., which provide both free and commercial versions of the software. Some of the functionality provided by SonPy requires a commercial version, for instance conversion of GDSII files to Sonnet project files.

The goal of SonPy is to replace any interactions between the user and the Sonnet GUI with Python functions, thereby allowing a fully automated circuit design and simulation process with for instance feedback loops based on the results of the simulations.

2.2 How can I use SonPy?

SonPy was developed with the following usage in mind:

1. The user has a GDSII file (.gds) with the basic circuit design, for instance produced in Python using gdspy. Designing the circuit in Python has the advantage that variables (resonator lengths, port attachment points ect.) can be used later in the script in relation with SonPy/Sonnet.
2. The GDSII file is converted to a Sonnet project file (.son) using SonPy. This requires a Sonnet license that allows gds conversion.
3. The Sonnet project is manipulated (metal properties are set, ports are added, frequency and parameter sweeps are set up ect.) using SonPy.
4. The user runs the Sonnet simulation through SonPy. This requires a Sonnet license that allows the simulation set up by the user (simple simulations without parameter sweeps are allowed in the free version.)
5. Data from the simulation is extracted.
6. Based on the simulation results we may go to step 1 and alter the circuit design.

Simple examples of usage are given in the example Python scripts available in the SonPy GitHub. Please have a look at these to understand the basic setup and usage.

2.3 Terminology

Although the user is assumed to be familiar with Sonnet and chip design lingo, we briefly go through some of the terminology used in Sonnet, SonPy and the Sonnet project file syntax manual [Son15]. Fig. 1 shows a screenshot from the Sonnet GUI highlighting some typical concepts.

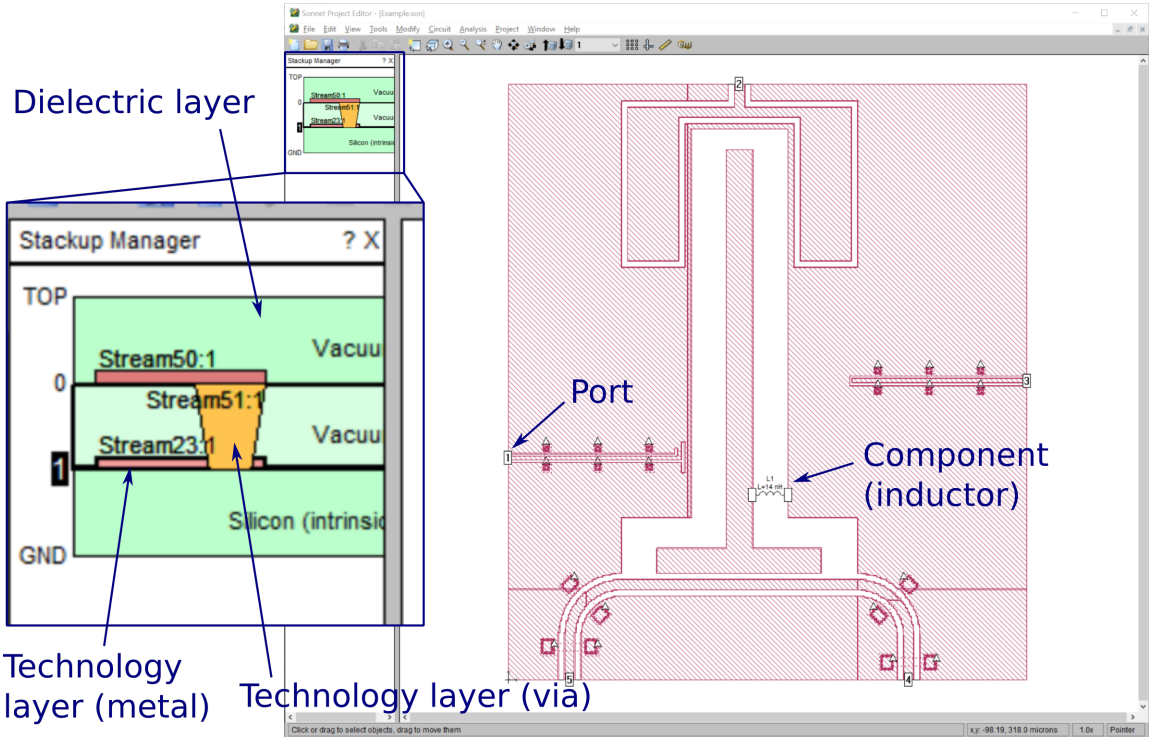

Fig. 1: Sonnet terminology illustrated in the Sonnet GUI.

Dielectric layer is abreviated *dlayer* in SonPy and indexed 0, 1, 2 ect. starting from the top. They are represented as green slabs in the Sonnet GUI, and are typically vacuum or Silicon. The dielectric layers are the fundamental layers of the system, and everything else (technnology layers, ports and components) reside in a dielectric layer. This means that removing a dielectric layer also removes any technology layers, ports and components associated with that dielectric layer.

Technology layer is abreviated *tlayer* in SonPy and is of type *metal*, *via* or *brick*. Metal tlayers are red in the Sonnet GUI and comprised of many polygons that make up most of the actual circuit design. Via tlayers are orange in the Sonnet GUI, and they are typically used to connect two metal layers, thus functioning as bridge pillars in an air bridge. Brick tlayers are blue in the Sonnet GUI and rarely used. Each tlayer is associated with the dlayer in which the tlayer is physically placed (via tlayers are associated to the dlayer in which is starts and also has a data slot for the dlayer it extends to). SonPy assumes that tlayers are mapped from a GDSII file (.gds file) and follow the default naming convention `StreamX:Y` with `X` the gds stream number and `Y` the gds object number. Tlayers are indexed by their gds stream number, i.e. this is the number to input to a SonPy function to modify a specific tlayer.

Ports are (usually) placed on the edge of the circuit/the bounding box. They are associated with a dlayer. Sonnet supports different types of ports, but only ports of *standard type* (STD) is implemented in SonPy. In the Sonnet GUI they are represented by a the port number in box.

Components can be placed on the circuit to model an ideal *inductor*, *resistor* or *capacitor*. Only components of type *ideal* are implemented in SonPy. Like ports they are associated with a dlayer. In the Sonnet GUI they are represented by a drawing illustrating the component type extending from one point to another. These two points are themselves ports (although not standard ports), but it means adding a components increases the number of ports by two.

The Sonnet GUI provides a nice overview of the stackup and any ports and components in the project. In SonPy the user can get a similar overview by calling the `printLayers()` function. For the project shown in Fig. 1 the function call `printLayers()` prints to the command line:

```
================== TOP ==================

  Dielectric layer:  0 (Vacuum)

================= LVL 0 =================

  Dielectric layer:  1 (Vacuum)
  Technology layer:  23 (METAL)
  Technology layer:  50 (METAL)
  Technology layer:  51 (VIA)
  Port:              1 (STD)
  Port:              2 (STD)
  Port:              3 (STD)
  Port:              4 (STD)
  Port:              5 (STD)
  Component:         "L1" (IND)

================= LVL 1 =================

  Dielectric layer:  2 (Silicon (intrinsic))

================== GND ==================
```

CHAPTER

THREE

DEVELOPER'S GUIDE

Note: This section can be skipped by regular users.

3.1 Sonnet project file syntax

To understand how SonPy is written it is crucial to understand the syntax of Sonnet project files (.son files) as explained in the manual [Son15]. Opening a Sonnet project file in a text editor makes it readable by humans. It is composed of a series of blocks that begins with a `BLOCKNAME` line and ends with a `BLOCKNAME END` line. Each block defines different properties of the Sonnet project, for instance the DIM block specifies the physical units, the GEO block sets the metal properties and defines all the polygons that make up the circuit, and the FREQ block sets the frequency sweep used in the simulation. Within each block are statements that defines every aspect of the project. SonPy does not implement all of the possibilities within Sonnet (that would be crazy!). Only those functionalities that where valuable to the EQuS Group at MIT at the time of development were implemented, but this guide should make it easier for you to implement your own features.

SonPy is comprised by classes and functions defined in `sonpy.py`. The functions fall into three categories:

1. Functions that runs a Sonnet subprogram (em.exe, emstatus.exe or gds.exe) found in Sonnet's program folder. They can be run from the command line, and SonPy runs them as a subprocess. These functions are used to convert a GDSII file to a Sonnet project file or to run the simulation.
2. Functions that manipulate the Sonnet project. They are used to add ports, change metal properties, set up the simulation ect.
3. Miscellaneous functions such as for reading or printing Sonnet project files (typically not necessary to run explicitly), reading output files from a Sonnet simulation, or printing information about the current project in the command line.

3.2 Python class structure for a Sonnet project

A Sonnet project in SonPy is an instance of the `sonnet` class. The instance variables of the `sonnet` class stores all the information needed for Python and Sonnet to interact, such as file paths to the Sonnet programs, Sonnet project file, GDSII file, data output file ect. The Sonnet project itself (equivalent to a Sonnet project file) is contained in the instance variable `project`, which is initialized as an instance of the `Project` class. The `Project` class mimics the structure of a Sonnet project file (compare with p. 8 of [Son15]):

```
class Project():
    # Only geometry projects are supported
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        self.preheader = None
        self.header = None
        self.dim = None
        self.geo = None
        self.control = None
        self.freq = None
        self.opt = None
        self.varswp = None
        self.fileout = None
        self.subdiv = None
        self.qsg = None
```

Each instance variable (except `preheader`) corresponds to a Sonnet project file block. They are all classes themselves containing stuff belonging to that block. For instance, for the GEO block SonPy has a class `Geo` intented to be stored as the `geo` instance variable in the `Project` class:

```
class Geo():
    def __init__(self):
        self.tmet = None
        self.bmet = None
        self.met = None
        self.box = None
        self.dlayers = []
        self.valvars = []
        self.lorgn = None
        self.npoly = None
```

Depending on the purpose, the instance variables of the `Geo` class may be a class, list of classes or a simply type like an integer. For instance, `box` is intended to be initialized as an instance of the `Box` class to store the data from the BOX statement (p. 33 of [Son15]):

```
class Box():
    def __init__(self):
        self.nlev = None
        self.xwidth = None
        self.ywidth = None
        self.xcells2 = None
        self.ycells2 = None
        self.nsubs = None
        self.eeff = None
```

Since each project only contains a single BOX statement, it is sufficient to only make room for one in the Python class. However, there may be any number of dielectric layers in the project, and so `dlayers` is intended to store a list of instances of the `Dlayer` class, each instance containing the data of a single layer. Lastly, some information can be saved as just a number, like `npoly` from the NUM statement (p. 43 of [Son15]) which specify the number of polygons in the project.

3.3 Reading a Sonnet project into SonPy

Given a Sonnet project file (like the output of a .gds to .son conversion), SonPy must read and store the Sonnet project. This is done with the function `readProject` which creates an instance of `Project` (called `project` stored in `self.project`) and carefully goes through the Sonnet project file line by line. The Sonnet project is then stored in the detailed class structure of `Project` and its instance variables. For instance, when the line `BEGIN GEO` is read in the Sonnet project file, the class `Geo` is initialized and assigned to `self.project.geo`. While looking through the GEO block we will for instance recognize the BOX statement (p. 33 of [Son15]) which defines some physical parameters of the circuit. SonPy has a corresponding `Box` class with the physical parameters as instance variables. The parameter names in SonPy (`nlev`, `xwidth`, `ywidth` ect.) are always inherited from the Sonnet project file syntax whenever possible. To see exactly which Sonnet project file parameters are stored as which SonPy parameters, you will have to look into the workings of the `readProject` function, and compare with the Sonnet project file syntax in [Son15]. An instance of `Box` is initialized with the parameters read from the Sonnet project file, and the `Box` instance is saved as `self.project.geo.box`. The BOX parameters can then be retrieved as `self.project.geo.box.nlev` for the integer parameters `nlev` which specify the number of layer levels in the project. In this way the Sonnet project is stored as a Russian doll, and every parameter of the project is accessible to SonPy.

Notice that the function `runGdsTranslator` also runs `readProject`, so when converting a GDSII file with `runGdsTranslator` the Sonnet project is automatically read into SonPy.

3.4 Manipulating the Sonnet project

When `readProject` has run, the entire Sonnet project is accessible to SonPy. Most of the functions in SonPy are dedicated to add, remove of alter stuff in the Sonnet project. For instance, the function `removeDlayer` removes a dielectric layer by removing the appropriate member from the list `self.project.geo.dlayers` of `Dlayer` class instances. The layer indices of the layers below the removed layer are updated (including everything that reside in those layers such as technology layers, ports and components). Finally the variable `nlev` which holds the number of dielectric layers is decresed by one with the line `self.project.geo.box.nlev += -1`. All manipulations of the Sonnet project is done by altering `Project` class instance `self.project`.

Before you add new functions of your own to SonPy, familiarize yourself with the code of the existing functions. Often the structures appear in different functions, for instance looping over all ports in all dielectric layers, or looping over all polygons in all technology layers in all dielectric layers, and you can straightforward copy or slightly alter existing code blocks. This also preserves consistency in coding style.

3.5 Updating the Sonnet project file

When all the appropriate changes had been made to the Sonnet project, and it is time to simulate the project, a new Sonnet project file must be created. This is done by the function `printProject` which overwrites the Sonnet project file with the modified project. It goes through all the data stored in `self.project` and writes the appropriate statements to the .son file following the Sonnet project file syntax.

Notice that the functions `runSimulation` and `runSimulationStatusMonitor` (both starting a Sonnet simulation) also calls `printProject`, so when running a simulation using these functions there is no need to explicitly run `printProject`.

CHAPTER

FOUR

API DOCUMENTATION

class sonpy.**sonnet**

Bases: `object`

Basic class for all interactions between Sonnet and Python, and for storing the Sonnet project. Start your interactions with SonPy by creating an instance of this class, like so:

```
>>> import sonpy
>>> snt = sonpy.sonnet()
```

All the functions of SonPy described in this API documention are functions defined in the *sonnet* class.

setSonnetInstallationPath(*path*)

Sets the path of the Sonnet installation.

Parameters **path** (*str*) – Path to Sonnet executables `em.exe`, `emstatus.exe` and `gds.exe`. Default is `C:\Program Files (x86)\Sonnet Software\14.54\bin\`.

setSonnetFile(*filename*)

Sets the filename of the Sonnet project file.

Parameters **filename** (*str*) – Sonnet project file. Default is `test.son`.

setSonnetFilePath(*path*)

Sets the path of the Sonnet project file.

Parameters **path** (*str*) – Path to the Sonnet project file. Default is `C:\Users\Lab\Desktop\sonnet_test\`.

setGdsFile(*filename*)

Sets the filename of the GDSII file. It furthermore sets the Sonnet project file and data file to the same name (but with extensions .son and .csv, respectively).

Parameters **filename** (*str*) – GDSII file. Default is `test.gds`.

setGdsFilePath(*path*)

Sets the path of the GDSII file. It furthermore sets the paths to the Sonnet project file and data file to the same path.

Parameters **path** (*str*) – Path to the GDSII file. Default is the Sonnet project path.

Example Say you have `myproject.gds` that you want to use that as a starting point for a Sonnet project.

```
>>> import os
>>> snt.setGdsFilePath(os.getcwd()) # get current work directory
>>> snt.setGdsFile('myproject.gds')
```

SonPy will now associate `myproject.son` and `myproject.csv` (in the current directory) with the current project, even if these files do not exist yet.

setDataFile(*filename*)

Sets the filename of the data file.

Parameters **filename** (*str*) – Data file. Default is the Sonnet project filename, but with the extension .csv instead of .son.

setDataFilePath(*path*)

Sets the filepath of the data file.

Parameters **path** (*str*) – Path to the data file. Default is the Sonnet project path.

setTemplateFile(*filename*)

Sets an existing Sonnet project file (in the current directory) as template. When converting a GDSII file, the resulting Sonnet project file will inherit the settings of the template.

Parameters **filename** (*str*) – Filename of the template Sonnet project file.

runGdsTranslator(*silent=False*)

Runs Sonnet's GDSII file to Sonnet project file translator. A Sonnet project file with the same name as the GDSII file is created in the same directory.

After the translation process has run, the following functions are called:

1. *readProject()*: Reads the created Sonnet project file into SonPy.
2. *collapseDlayers()*: Removes empty dielectric layers.
3. *cropBox()*: Crops the bounding box to the edge of the circuit.

Parameters **silent** (*bool*) – Toggle wait message.

readProject()

Reads the Sonnet project file into SonPy. This function is run in *runGdsTranslator()* to ensure the created Sonnet project file is read into SonPy for further manipulation.

printLayers()

Prints the layer configuration of the project to the command prompt. For each dielectric layer the following is printed:

```
Dielectric layer:  dlayer_index (name)
Technology layer:  tlayer_index (tlayer_type)
Port:              portnum (port_type)
Component:         name (component_type)
```

printParameters()

Prints the defined variable parameters and sweeps set in the project to the command prompt. The frequency/parameter sweeps printed out will run in the Sonnet simulation, and data will be written to the set data file. Variables are printed in following form:

```
varname (unittype)
```

printProject()

Prints (overwrites) the Sonnet project with the changes made in SonPy to the Sonnet project file. This function runs before the Sonnet simulation to ensure any changes made in SonPy are recorded in the Sonnet project file Sonnet's simulation software reads.

cropBox(*xcellsize=1*, *ycellsize=1*)

Crops the bounding box (used in Sonnet to confine the simulation space) to the circuit of the circuit.

Parameters

- **xcellsize** (*float*) – Cellsize in x direction.
- **ycellsize** (*float*) – Cellsize in y direction.

addPort(*xcoord*, *ycoord*, *xmargin=0.005*, *ymargin=0.005*, ***kwargs*)

Adds a port (of standard type). Since Sonnet's GDSII translator ever so slightly shifts the coordinates used in the GDSII file it is often necessary to look for attachment points with a small margin. For instance, say you originally planned to add a port at the edge of your circuit at (0, 100). After the GDSII file has been translated into a Sonnet project file, this points has shifted to, say, (0, 99.997). Trying to add the port at (0, 100) will throw an error because this point is no longer at the edge of the circuit (or any of the polygons). Looking for possible attachment points (i.e. polygon edges) with a small margin will find to correct point (0, 99.997).

Parameters

- **xcoord** (*float*) – Attachment x coordinate.
- **ycoord** (*float*) – Attachment y coordinate.
- **xmargin** (*float*) – Margin in x direction.
- **ymargin** (*float*) – Margin in y direction.

Keyword arguments:

Parameters

- **resist** (*float*) – Port parameter defined in [Son15] under POR1.
- **react** (*float*) – Port parameter defined in [Son15] under POR1.
- **induct** (*float*) – Port parameter defined in [Son15] under POR1.
- **capac** (*float*) – Port parameter defined in [Son15] under POR1.
- **tlayer_index** (*int or list of ints*) – Restrict the search for attachment points to a single or list of technology layers (useful if several technology layers have overlapping polygon edges at the attachment point).

addComponent(*x1*, *y1*, *x2*, *y2*, *tlayer_index*, *component_type='ind'*, *value=10*, *xmargin=0.005*, *ymargin=0.005*, ***kwargs*)

Adds an ideal component. Attachment point margins are the same as for *addPort()*.

Parameters

- **x1** (*float*) – Attachment x coordinate for the first port.
- **y1** (*float*) – Attachment y coordinate for the first port.
- **x2** (*float*) – Attachment x coordinate for the second port.
- **y2** (*float*) – Attachment y coordinate for the second port.
- **xmargin** (*float*) – Margin in x direction.
- **ymargin** (*float*) – Margin in y direction.
- **tlayer_index** (*int*) – Index (gds stream number) of the technology layer the component will live in.

- **component_type** (*str*) – Type of component. Should be "ind" (inductor), "cap" (capacitor) or "res" (resistor).
- **value** (*float*) – Value of the component in units suitable for the component type.

Keyword arguments:

Parameters

- **name** (*str*) – Name for the component. Default is L, C, R followed by a number for inductors, capacitors and resistors. For instance, the first ideal capacitor is named "L1", the next "L2" ect.
- **smdp1_portnum** (*int*) – Port number for first port, see [Son15] under SMD. Default is the number of existing ports + 1.
- **smdp1_pinnum** (*int*) – See [Son15] under SMD.
- **smdp2_portnum** (*int*) – Port number for second port, see [Son15] under SMD. Default is the number of existing ports + 2.
- **smdp2_pinnum** (*int*) – See [Son15] under SMD.

addPoly(*coord_list*, *lay_name*, *inherit='INH'*, *ilevel=0*, *mtype=0*, *filltype='N'*, *debugid=0*, *xmin=1*, *ymin=1*, *xmax=100*, *ymax=100*, *conmax=0*, *edgemesh='Y'*)

Adds a polygon to the layer lay_name with the coordinates given by coord_list. See [Son15] for a detailed description of the parameters.

Parameters

- **coord_list** (*array*) – Coordinate list, e.g. [(5, 5), (6, 5), (6, 6), (5, 6), (5, 5)]
- **lay_name** (*str*) – Name of the layer to put the polygon on

removeDlayer(*dlayer_index=0*)

Removes a dielectric layer including any technology layers, ports or components that reside in the layer. Any via technology layers that extend to this dielectric layer are also removed.

Parameters **dlayer_index** (*int*) – Index of the layer that will be removed.

removeEmptyDlayers()

Removes all dielectric layers that do not contain any technology layers except for the bottom dielectric layer, which should always be empty.

collapseDlayers()

Sends all technology layers, ports and components to the top dielectric layer (dlayer_index = 0) and removes all dielectric layers except for the two top layers.

This function is run after the GDSII translator for the following reason. Say the GDSII has a layer with gds stream number 23. This layer will become a technology layer in a dielectric layer of index 23, so a lot of empty dielectric layers are created in this process. This function removes these layers. After this operation there is an empty bottom dielctric layer (as there should be) and a single dielctric layer on top.

addDlayer(*dlayer_index=0*, ***kwargs*)

Adds a dielectric layer to the project.

Parameters **dlayer_index** (*int*) – Index of the dielctric layer after the addition. By default the new layer is added to the top.

Keyword arguments:

Parameters

- **thickness** (*float*) – See [Son15] under BOX. Default is 0.

- **erel** (*float*) – See [Son15] under BOX. Default is 1.
- **mrel** (*float*) – See [Son15] under BOX. Default is 1.
- **eloss** (*float*) – See [Son15] under BOX. Default is 0.
- **mloss** (*float*) – See [Son15] under BOX. Default is 0.
- **esignma** (*float*) – See [Son15] under BOX. Default is 0.
- **name** (*str*) – See [Son15] under BOX. Default is "New layer".

setDlayer(*dlayer_index*, ***kwargs*)

Sets the parameters of a dielectric layer to those specified by the keyword arguments. Only parameters given as keyword arguments are changed.

Parameters **dlayer_index** (*int*) – Index of the dielectric layer to modify.

Keyword arguments:

Parameters

- **thickness** (*float*) – See [Son15] under BOX.
- **erel** (*float*) – See [Son15] under BOX.
- **mrel** (*float*) – See [Son15] under BOX.
- **eloss** (*float*) – See [Son15] under BOX.
- **mloss** (*float*) – See [Son15] under BOX.
- **esignma** (*float*) – See [Son15] under BOX.
- **name** (*str*) – See [Son15] under BOX.

setTlayer(*tlayer_index*, ***kwargs*)

Sets the parameters of a technology layer to those specified by the keyword arguments. Only parameters given as keyword arguments are changed.

Parameters **tlayer_index** (*int*) – Index (gds stream number) of the technnology layer to modify.

Keyword arguments:

Parameters

- **dlayer_index** (*int*) – Index of the dielctric layer in which the technology layer reside.
- **to_dlayer_index** (*int*) – Index of the dielectric layer a via technology layer extends to. Only for via technology layers.
- **tlayer_type** (*str*) – The type of technology layer. Should be "metal", "via" or "brick". If a non-via layer is changed to via a value of to_dlayer_index should also be given.
- **name** (*str*) – Name of the technology layer.
- **brick_name** (*str*) – Name of the brick material. See [Son15] under BRI and BRA. Required if a brick tlayer is defined.
- **lossless** (*bool*) – Toggle for lossless metal.
- **filltype** (*str*) – Polygon filltype. Either "N" (staircase fill), "T" (diagonal fill) or "V" (conformal mesh). See [Son15] under NUM.
- **edgemesh** (*str*) – Either "Y" (on) or "N" (off). See [Son15] under NUM.

- **meshingfill** (*str*) – Either "RING", "CENTER", "VERTICES", "SOLID" or "BAR". See [Son15] under NUM.
- **pads** (*str*) – Either "NOCOVERS" or "COVERS". See [Son15] under NUM.

addBrick(*erel=1*, *loss_tan=0*, *cond=0*, *name='Air'*)

Adds a dielectric layer to the project.

Parameters

- **erel** (*float or list of floats of length 3*) – relative permitivity or [erelx, erely, erelz]
- **loss_tan** (*float or list of floats of length 3*) – loss tangent or [erelx, erely, erelz]
- **cond** (*float or list of floats of length 3*) – conductivity or [erelx, erely, erelz]

setFrequencySweep(*f1=5*, *f2=8*, *fstep=None*)

Sets the frequency sweep to be run during the simulation.

Parameters

- **f1** (*float*) – Minimum frequency in the sweep.
- **f2** (*float*) – Maximum frequency in the sweep.
- **fstep** (*None or float*) – The stepsize in a linear frequency sweep. If set to None an adaptive frequency sweep will run.

addParameter(*parameter*, *unittype=None*, ***kwargs*)

Adds a variable parameter to the project.

Parameters

- **parameter** (*str*) – Name of the parameter. If the name of an existing ideal component is given, the variable will be associated to that component's value.
- **unittype** (*str*) – The unit type of the variable. See [Son15] under VALVAR. If parameter is the name of an existing ideal component, the unit type will be inherited from the component.

Keyword arguments:

Parameters

- **value** (*float*) – The value of the parameter. Default is 30. If parameter is the name of an existing component, the value will be inherited from the component.
- **description** (*str*) – Description of the parameter. Default is empty quotes, "".

addParameterSweep(*parameter*, *pmin*, *pmax*, *pstep*, ***kwargs*)

Adds a parameter sweep to the project. Unless the keyword argument to_existing_sweep is given any previously set parameter sweeps will be overwritten. You can see the active sweeps set up in the project by running *printParameters()*. The parameter sweep runs within a given frequency sweep, either previously set with *setFrequencySweep()* or given with keyword arguments. If no frequency sweep has been defined for the project, the default from *setFrequencySweep()* will be set.

Parameters

- **parameter** (*str*) – Name of the parameter to sweep. It must be the name of an existing parameter, for instance defined by *addParameter()*.
- **pmin** (*float*) – Minimum parameter value in the sweep.

- **pmax** (*float*) – Maximum parameter value in the sweep.
- **pstep** (*None or float*) – The stepsize in a linear sweep. If set to None an adaptive sweep will run.

Keyword arguments:

Parameters

- **to_existing_sweep** (*int*) – Index of an existing parameter sweep (1, 2, ect.). If given the parameter sweep is added to the existing parameter sweep such that several parameters will be sweept during the simulation.
- **ytype** (*str*) – Type of sweep. Either "N", "Y", "YN", "YC", "YS", or "YE". See [Son15] under VARSWP.
- **f1** (*float*) – Minimum frequency in the sweep.
- **f2** (*float*) – Maximum frequency in the sweep.
- **fstep** (*None or float*) – The stepsize in a linear frequency sweep. If set to None an adaptive frequency sweep will run.

setOutput(***kwargs*)

Sets an output file for simulation data. By default a single spreadsheet file (.csv) is created with the same filename as the Sonnet project file. By default the data format is S-parameters given in dB and angles. See [Son15] under FILEOUT for other options.

Keyword arguments:

Parameters

- **filetype** (*str*) – See [Son15] under FILEOUT. Default is "CSV".
- **embed** (*str*) – See [Son15] under FILEOUT. Default is "D".
- **abs_inc** (*str*) – See [Son15] under FILEOUT. Default is "Y".
- **filename** (*str*) – See [Son15] under FILEOUT. Default is "$BASENAME.csv".
- **comments** (*str*) – See [Son15] under FILEOUT. Default is "NC".
- **sig** (*int*) – See [Son15] under FILEOUT. Default is 8.
- **partype** (*str*) – See [Son15] under FILEOUT. Default is "S".
- **parform** (*str*) – See [Son15] under FILEOUT. Default is "DB".
- **ports** (*str*) – See [Son15] under FILEOUT. Default is "R 50".
- **folder** (*str*) – See [Son15] under FOLDER. Default is None.

getOutput(*data='frequency'*, *run=1*)

Returns a list of data from the data resulting created after a Sonnet simulation. It is assumed that the data file is a .csv file, which is the default setting when running *setOutput()*. This function makes it easy to extract simulation data and plot it within Python.

Parameters

- **data** (*str*) – String specifying the data. It must follow the naming in the .csv file, for instance "MAG[S12]" for the magnitude of the S12-parameter. The default is frequency data.
- **run** (*int*) – The sweep number if a parameter was swept during the simulation.

Returns List of data.

runSimulation()

Runs the Sonnet simulation without the pop-up status monitor. This function also calls *printProject()*, thereby saving any changes made in SonPy to the Sonnet project file before starting the simulation.

runSimulationStatusMonitor()

Runs the Sonnet simulation with the pop-up status monitor. This function also calls *printProject()*, thereby saving any changes made in SonPy to the Sonnet project file before starting the simulation.

addComment(*string*)

Adds a comment to the top of the Sonnet project file.

Parameters **string** (*str*) – Comment.

runMakeover()

Applies a series of other functions. This makeover function takes a GDSII file, runs it through a series of standardized tasks such that the project after the makeover is ready for adding ports, applying other specific settings and simulation.

The following operations are applied:

1. *runGdsTranslator()*: Translate the GDSII file and read the project into SonPy.
2. Set the bottom dielectric layer to Silicon with the appropriate parameters.

Depending on the number of technology layers, we either create an air bridge, or we do not.

3. If there is only one technology layer (assumed to have gds stream 23), there is no air bridge, and the top layer is set to vacuum. The technology layer is set to lossless metal.

3. If there are three technology layers (assumed to have gds streams 23, 50 and 51), we create an air bridge from 50 (the top) and 51 (set to a via for bridge pillars). All technology layers are set to lossless metal. The top dielectric layer is made a thin vacuum at the bridge pillar level, and a new thick vacuum layer is added on top of the substrate.
4. *setFrequencySweep()*: The default frequency sweep is set.
5. *setOutput()*: The default output data file is set.

BIBLIOGRAPHY

[Son15] Sonnet Project Format, Release 16, Sonnet Software Inc. (2018). Can be found here.

PYTHON MODULE INDEX

S

INDEX