



# scikit-**rf**

Object Oriented RF Engineering

## **scikit-**rf** Documentation**

***Release 0.13***

**Alex Arsenovic**

January 28, 2013



# CONTENTS

<b>1</b>	<b>Tutorials</b>	<b>3</b>
1.1	Installation . . . . .	3
1.2	Introduction . . . . .	4
1.3	Networks . . . . .	11
1.4	Plotting . . . . .	20
1.5	NetworkSet . . . . .	30
1.6	Virtual Instruments . . . . .	35
1.7	Calibration . . . . .	36
1.8	Media . . . . .	40
<b>2</b>	<b>Examples</b>	<b>47</b>
2.1	Visualizing a Single Stub Matching Network . . . . .	47
2.2	One-Port Calibration . . . . .	51
<b>3</b>	<b>Reference</b>	<b>53</b>
3.1	frequency (skrf.frequency) . . . . .	53
3.2	network (skrf.network) . . . . .	57
3.3	networkSet (skrf.networkSet) . . . . .	132
3.4	plotting (skrf.plotting) . . . . .	138
3.5	mathFunctions (skrf.mathFunctions) . . . . .	144
3.6	tlineFunctions (skrf.tlineFunctions) . . . . .	147
3.7	constants (skrf.constants) . . . . .	155
3.8	util (skrf.util) . . . . .	156
3.9	io (skrf.io) . . . . .	158
3.10	calibration (skrf.calibration) . . . . .	166
3.11	media (skrf.media) . . . . .	185
3.12	Indices and tables . . . . .	255
	<b>Python Module Index</b>	<b>257</b>
	<b>Index</b>	<b>259</b>



This documentation is also available in pdf form: [scikit-\*\*rf\*\*.pdf](#)



# TUTORIALS

## 1.1 Installation

### Contents

- Installation
  - Introduction
  - **skrf** Installation
  - Testing Installation
  - Requirements
    - \* Debian-Based Linux
    - \* Necessary
    - \* Optional

### 1.1.1 Introduction

The requirements to run **skrf** are basically a [Python](#) environment setup to do numerical/scientific computing. If you are new to development, you may want to install a pre-built scientific python IDE like [pythonxy](#) or the [enthought python distribution](#). Either of these *distributions* will install all requirements, as well as provide a nice environment to get started in. If you dont want use [pythonxy](#) or [enthought](#) see [Requirements](#).

---

**Note:** If you want to use **skrf** for instrument control you will need to install [pyvisa](#) as well as the NI-GPIB drivers. You may also be interested in [Pythics](#) , which provides a simple way to build graphical interfaces to virtual instruments.

---

### 1.1.2 skrf Installation

Once the requirements are installed, there are two choices for installing **skrf**:

- windows installer
- python source package

These can be found at <http://scikit-rf.org/download.html>

If you dont know how to install a python module and dont care to learn how, you want the windows installer.

The current version can be accessed through [github](#). This is mainly of interest for developers.

### 1.1.3 Testing Installation

If import `skrf` and dont recieve an error, then installation was succesful.

```
In [1]: import skrf as rf
```

If instead you get an error like this,

```
In [2]: import skrf as rf
```

```
-----  
ImportError                                Traceback (most recent call last)  
<ipython-input-1-41c4ee663aa9> in <module>()  
----> 1 import skrf as rf  
\  
ImportError: No module named skrf
```

Then installation was unsuccessful. If you need help post to the [mailing list](#).

### 1.1.4 Requirements

#### Debian-Based Linux

For debian-based linux users who dont want to install `pythonxy`, here is a one-shot line to install all requirements,

```
sudo apt-get install python-pyvisa python-numpy python-scipy python-matplotlib ipython python python-
```

Once `setuptools` is installed you can install `skrf` through `easy_install`

```
easy_install scikit-rf
```

#### Necessary

- python ( $\geq 2.6$ ) <http://www.python.org/>
- numpy <http://numpy.scipy.org/>
- scipy <http://www.scipy.org/>
- matplotlib <http://matplotlib.sourceforge.net/>

#### Optional

- ipython <http://ipython.scipy.org/moin/> - for interactive shell
- pyvisa <http://pyvisa.sourceforge.net/pyvisa/> - for instrument control
- Pythics <http://code.google.com/p/pythics> - instrument control and gui creation

## 1.2 Introduction



**Contents**

- Introduction
  - Introduction
  - Networks
    - \* Linear Operations
    - \* Cascading and De-embedding
  - Plotting
  - NetworkSet
    - \* Statistical Properties
    - \* Plotting Uncertainty Bounds
  - Virtual Instruments
  - Calibration
    - \* One Port Calibration
  - Media
    - \* Media Types
    - \* Network Components

## 1.2.1 Introduction

This is a brief introduction to **skrf** which highlights a range of features without going into detail on any single one. At the end of each section there are links to other tutorials, that provide more information about a given feature. The intended audience are those who have a working python stack, and are somewhat familiar with python. If you are unfamiliar with python, please see [scipy's Getting Started](#) .

Although not essential, these tutorials are most easily followed by using the `ipython` shell with the `--pylab` flag.

```
> ipython --pylab
In [1]:
```

Using `ipython` with the `pylab` flag imports several commonly used functions, and turns on `interactive plotting mode` which causes plots to display immediately.

Throughout this tutorial, and the rest of the scikit-rf documentation, it is assumed that **skrf** has been imported as `rf`. Whether or not you follow this convention in your own code is up to you.

```
In [1]: import skrf as rf
```

If this produces an error, please see [Installation](#).

---

**Note:** The example code in these tutorials make use of files that are distributed with the source package. The working directory for these code snippets is `scikit-rf/doc/`, hence all data files are referenced relative to that directory. If you do not have the source package, then you may access these files through the `skrf.data` module (ie `from skrf.data import ring_slot`)

---

## 1.2.2 Networks

The `Network` object represents a N-port microwave `Network`. A `Network` can be created in a number of ways. One way is from data stored in a touchstone file.

```
In [1]: ring_slot = rf.Network('../skrf/data/ring_slot.s2p')
```

A short description of the network will be printed out if entered onto the command line

```
In [1]: ring_slot
Out[1]: 2-Port Network: 'ring_slot', 75-110 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j]
```

The basic attributes of a microwave `Network` are provided by the following properties :

- `Network.s` : Scattering Parameter matrix.
- `Network.z0` : Port Characteristic Impedance matrix.
- `Network.frequency` : Frequency Object.

All of the network parameters are complex `numpy.ndarray`'s of shape  $F \times N \times N$ , where  $F$  is the number of frequency points and  $N$  is the number of ports. The `Network` object has numerous other properties and methods which can be found in the `Network` docstring. If you are using IPython, then these properties and methods can be 'tabbed' out on the command line.

```
In [1]: short.s<TAB>
rf.data.line.s          rf.data.line.s_arcl          rf.data.line.s_im
rf.data.line.s11       rf.data.line.s_arcl_unwrap  rf.data.line.s_mag
...
```

## Linear Operations

Element-wise mathematical operations on the scattering parameter matrices are accessible through overloaded operators. To illustrate their usage, load a couple `Network`s stored in the `data` module.

```
In [1]: short = rf.data.wr2p2_short
```

```
In [2]: delayshort = rf.data.wr2p2_delayshort
```

```
In [3]: short - delayshort
```

```
Out[3]: 1-Port Network: 'wr2p2,short', 330-500 GHz, 201 pts, z0=[ 50.+0.j]
```

```
In [4]: short + delayshort
```

```
Out[4]: 1-Port Network: 'wr2p2,short', 330-500 GHz, 201 pts, z0=[ 50.+0.j]
```

## Cascading and De-embedding

Cascading and de-embedding 2-port `Network`s can also be done through operators. The `cascade()` function can be called through the power operator, `**`. To calculate a new network which is the cascaded connection of the two individual `Network`s `line` and `short`,

```
In [1]: short = rf.data.wr2p2_short
```

```
In [2]: line = rf.data.wr2p2_line
```

```
In [3]: delayshort = line ** short
```

De-embedding can be accomplished by cascading the *inverse* of a network. The inverse of a network is accessed through the property `Network.inv`. To de-embed the `short` from `delay_short`,

```
In [1]: short = line.inv ** delayshort
```

For more information on the functionality provided by the `Network` object, such as interpolation, stitching, n-port connections, and IO support see the [Networks](#) tutorial.

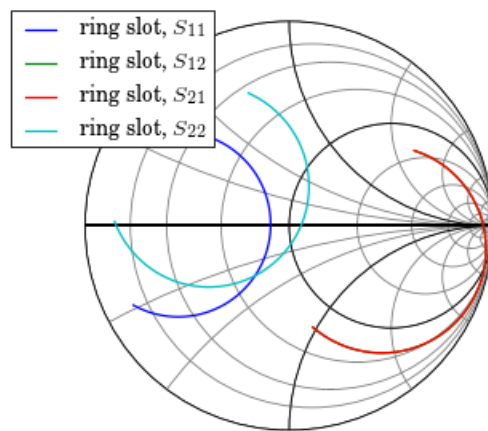
### 1.2.3 Plotting

Amongst other things, the methods of the `Network` class provide convenient ways to plot components of the network parameters,

- `Network.plot_s_db()` : plot magnitude of s-parameters in log scale
- `Network.plot_s_deg()` : plot phase of s-parameters in degrees
- `Network.plot_s_smith()` : plot complex s-parameters on Smith Chart
- ...

To plot all four s-parameters of the `ring_slot` on the Smith Chart.

```
In [1]: ring_slot.plot_s_smith();
```



For more detailed information about plotting see the [Plotting](#) tutorial

### 1.2.4 NetworkSet

The `NetworkSet` object represents an unordered set of networks and provides methods for calculating statistical quantities and displaying uncertainty bounds.

A `NetworkSet` is created from a list or dict of `Network`'s. This can be done quickly with `read_all()`, which loads all skrf-readable objects in a directory. The argument `contains` is used to load only files which match a given substring.

```
In [1]: rf.read_all('../skrf/data/', contains='ro')
Out[1]:
{'ro,1': 1-Port Network: 'ro,1', 500-750 GHz, 201 pts, z0=[ 50.+0.j],
 'ro,2': 1-Port Network: 'ro,2', 500-750 GHz, 201 pts, z0=[ 50.+0.j],
 'ro,3': 1-Port Network: 'ro,3', 500-750 GHz, 201 pts, z0=[ 50.+0.j]}
```

This can be passed directly to the `NetworkSet` constructor,

```
In [1]: ro_dict = rf.read_all('../skrf/data/', contains='ro')
In [2]: ro_ns = rf.NetworkSet(ro_dict, name='ro set') #name is optional
```

```
In [3]: ro_ns
Out[3]: A NetworkSet of length 3
```

## Statistical Properties

Statistical quantities can be calculated by accessing properties of the `NetworkSet`. For example, to calculate the complex average of the set, access the `mean_s` property

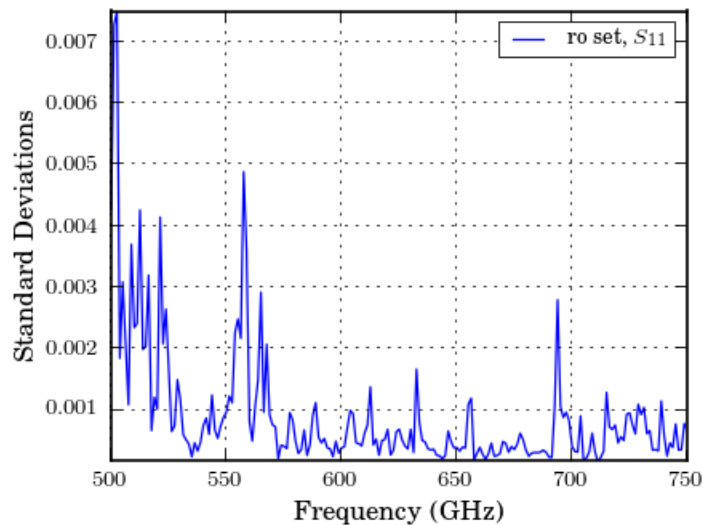
```
In [1]: ro_ns.mean_s
Out[1]: 1-Port Network: 'ro set', 500-750 GHz, 201 pts, z0=[ 50.+0.j]
```

Similarly, to calculate the complex standard deviation of the set,

```
In [1]: ro_ns.std_s
Out[1]: 1-Port Network: 'ro set', 500-750 GHz, 201 pts, z0=[ 50.+0.j]
```

These methods return a `Network` object, so the results can be saved or plotted in the same way as you would with a `Network`. To plot the magnitude of the standard deviation of the set,

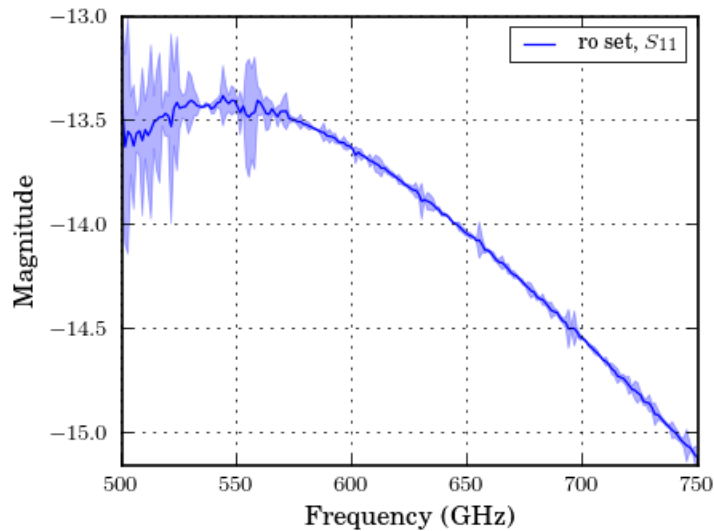
```
In [1]: figure();
In [2]: ro_ns.std_s.plot_s_re(y_label='Standard Deviations')
```



## Plotting Uncertainty Bounds

Uncertainty bounds on any network parameter can be plotted through the methods

```
In [1]: figure();
In [2]: ro_ns.plot_uncertainty_bounds_s_db()
```



See the *NetworkSet* tutorial for more information.

## 1.2.5 Virtual Instruments

**Warning:** The `vi` module is not well written or tested at this point.

The `vi` module holds classes for GPIB/VISA instruments that are intricately related to `skrf`, ie mostly VNA's. The VNA classes were created for the sole purpose of retrieving data so that calibration and measurements could be carried out offline by `skrf`, control of most other VNA capabilities is neglected.

**Note:** To use the virtual instrument classes you must have `pyvisa` installed.

A list of VNA's that have been are partially supported.

- HP8510C
- HP8720
- PNAX
- ZVA40

An example usage of the HP8510C class to retrieve some s-parameter data

```
In [1]: from skrf.vi import vna
```

```
In [2]: my_vna = vna.HP8510C(address =16)
```

*#if an error is thrown at this point there is most likely a problem with your visa setup*

```
In [3]: dut_1 = my_vna.s11
```

```
In [4]: dut_2 = my_vna.s21
```

```
In [5]: dut_3 = my_vna.two_port
```

Unfortunately, the syntax is different for every VNA class, so the above example wont directly translate to other VNA's. Re-writing all of the VNA classes to follow the same convention is on the [TODO list](#)

## 1.2.6 Calibration

**skrf** has support for one and two-port calibration. **skrf**'s default calibration algorithms are generic in that they will work with any set of standards. If you supply more calibration standards than is needed, **skrf** will implement a simple least-squares solution. **skrf** does not currently support TRL.

Calibrations are performed through a `Calibration` class. Creating a `Calibration` object requires at least two pieces of information:

- a list of measured `Network`'s
- a list of ideal `Network`'s

The `Network` elements in each list must all be similar (same #ports, frequency info, etc) and must be aligned to each other, meaning the first element of ideals list must correspond to the first element of measured list.

Optionally, other information can be provided when relevant such as,

- calibration algorithm
- enforce reciprocity of embedding networks
- etc

When this information is not provided **skrf** will determine it through inspection, or use a default value.

Below is an example script illustrating how to create a `Calibration`. See the *Calibration* tutorial for more details and examples.

### One Port Calibration

This example is the same as the first except more concise.

```
import skrf as rf

my_ideals = rf.read_all('ideals/')
my_measured = rf.read_all('measured/')
duts = rf.read_all('measured/')

## create a Calibration instance
cal = rf.Calibration(\
    ideals = [my_ideals[k] for k in ['short', 'open', 'load']],
    measured = [my_measured[k] for k in ['short', 'open', 'load']],
)

caled_duts = [cal.apply_cal(dut) for dut in duts.values()]
```

## 1.2.7 Media

**skrf** supports the microwave network synthesis based on transmission line models. Network creation is accomplished through methods of the `Media` class, which represents a transmission line object for a given medium. Once constructed, a `Media` object contains the necessary properties such as propagation constant and characteristic impedance, that are needed to generate microwave circuits.

The basic usage looks something like this,

```
In [1]: import skrf as rf

In [2]: freq = rf.Frequency(75,110,101,'ghz')
```

```
In [3]: cpw = rf.media.CPW(freq, w=10e-6, s=5e-6, ep_r=10.6)
```

```
In [4]: cpw.line(100*1e-6, name = '100um line')
```

```
Out[4]: 2-Port Network: '100um line', 75-110 GHz, 101 pts, z0=[ 50.06074662+0.j 50.06074662+0.j]
```

**Warning:** The network creation and connection syntax of `skrf` are cumbersome if you need to doing complex circuit design. For a this type of application, you may be interested in using `QUCS` instead. `skrf`'s synthesis capabilities lend themselves more to scripted applications such as *Design Optimization* or batch processing.

## Media Types

Specific instances of Media objects can be created from relevant physical and electrical properties. Below is a list of mediums types supported by `skrf`,

- `CPW`
- `RectangularWaveguide`
- `Freespace`
- `DistributedCircuit`
- `Media`

## Network Components

Here is a brief list of some generic network components `skrf` supports,

- `match()`
- `short()`
- `open()`
- `load()`
- `line()`
- `thru()`
- `tee()`
- `delay_short()`
- `shunt_delay_open()`

Usage of these methods can is demonstrated below.

To create a 1-port network for a coplanar waveguide short (this neglects discontinuity effects),

```
In [1]: cpw.short(name = 'short')
```

```
Out[1]: 1-Port Network: 'short', 75-110 GHz, 101 pts, z0=[ 50.06074662+0.j]
```

Or to create a 90° section of cpw line,

```
In [1]: cpw.line(d=90, unit='deg', name='line')
```

```
Out[1]: 2-Port Network: 'line', 75-110 GHz, 101 pts, z0=[ 50.06074662+0.j 50.06074662+0.j]
```

See [Media](#) for more information about the Media object and network creation.

## 1.3 Networks

**Contents**

- Networks
  - Introduction
  - Creating Networks
  - Network Basics
  - Network Operators
    - \* Linear Operations
    - \* Cascading and De-embedding
  - Connecting Multi-ports
  - Interpolation and Stitching
  - Reading and Writing
  - Impedance and Admittance Parameters
  - Creating Networks ‘From Scratch’
  - Sub-Networks
  - References

### 1.3.1 Introduction

For this tutorial, and the rest of the scikit-rf documentation, it is assumed that `skrf` has been imported as `rf`. Whether or not you follow this convention in your own code is up to you.

```
In [1]: import skrf as rf
```

If this produces an error, please see *Installation*. The code in this tutorial assumes that you are in the directory `scikit-rf/doc`.

### 1.3.2 Creating Networks

`skrf` provides an object for a N-port microwave `Network`. A `Network` can be created in a number of ways. One way is from data stored in a touchstone file.

```
In [1]: ring_slot = rf.Network('../skrf/data/ring_slot.s2p')
```

A short description of the network will be printed out if entered onto the command line

```
In [1]: ring_slot
Out[1]: 2-Port Network: 'ring_slot', 75-110 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j]
```

Networks can also be created from a pickled Network (written by `Network.write()`),

```
In [1]: ring_slot = rf.Network('../skrf/data/ring_slot.ntwk')
```

or from directly passing values for the frequency, s-paramters and `z0`.

```
In [1]: custom_ntwk = rf.Network(f = [1,2,3], s = [-1, 1j, 0], z0=50)
```

See `Network.__init__()` for more information on network creation.

### 1.3.3 Network Basics

The basic attributes of a microwave `Network` are provided by the following properties :

- `Network.s` : Scattering Parameter matrix.



- `Network.z0` : Port Characteristic Impedance matrix.
- `Network.frequency` : Frequency Object.

All of the network parameters are represented internally as complex `numpy.ndarray` 's of shape  $F \times N \times N$ , where  $F$  is the number of frequency points and  $N$  is the number of ports.

```
In [1]: shape(ring_slot.s)
Out[1]: (201, 2, 2)
```

Note that the indexing starts at 0, so the first 10 values of  $S_{11}$  can be accessed with

```
In [1]: ring_slot.s[:10,0,0]
Out[1]:
array([-0.50372318+0.45784448j , -0.49581904+0.45707698j,
       -0.48782538+0.4561578j , -0.47974451+0.45508186j,
       -0.47157898+0.45384372j, -0.46333160+0.45243787j,
       -0.45500548+0.45085878j, -0.44660400+0.44910088j,
       -0.43813086+0.4471586j , -0.42959005+0.44502637j])
```

The `Network` object has numerous other properties and methods which can found in the `Network` docstring. If you are using IPython, then these properties and methods can be 'tabbed' out on the command line.

```
In [1]: short.s<TAB>
rf.data.line.s          rf.data.line.s_arcl          rf.data.line.s_im
rf.data.line.s11       rf.data.line.s_arcl_unwrap rf.data.line.s_mag
...
```

---

**Note:** Although this tutorial focuses on s-parametes, other network representations such as Impedance (`Network.z`) and Admittance Parameters (`Network.y`) are available as well, see [Impedance and Admittance Parameters](#) .

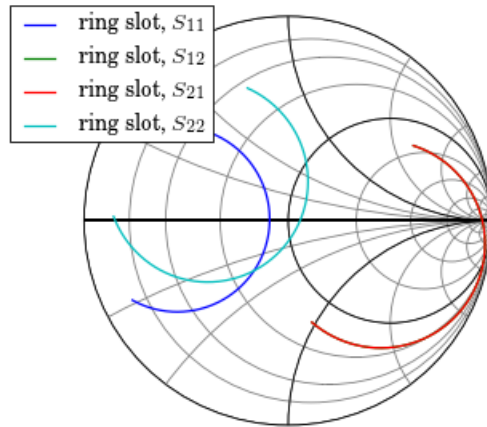
---

Amongst other things, the methods of the `Network` class provide convenient ways to plot components of the network parameters,

- `Network.plot_s_db()` : plot magnitude of s-parameters in log scale
- `Network.plot_s_deg()` : plot phase of s-parameters in degrees
- `Network.plot_s_smith()` : plot complex s-parameters on Smith Chart
- ...

To plot all four s-parameters of the `ring_slot` on the Smith Chart.

```
In [1]: ring_slot.plot_s_smith();
```

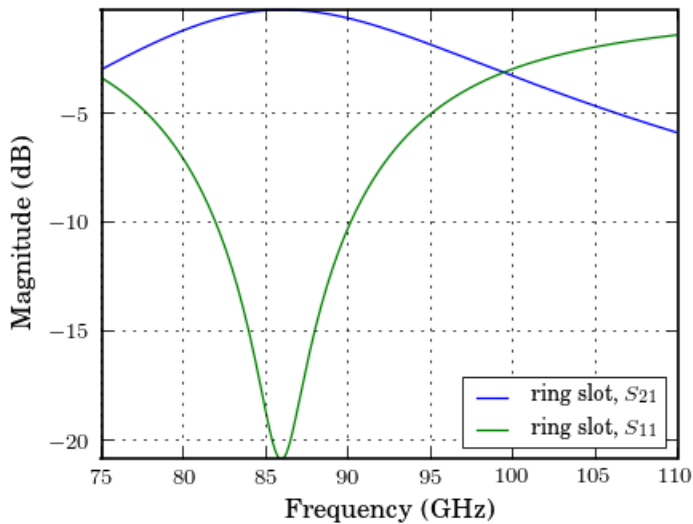


Or plot a pair of s-parameters individually, in log magnitude

```
In [1]: figure();
```

```
In [2]: ring_slot.plot_s_db(m=1, n=0); # s21
```

```
In [3]: ring_slot.plot_s_db(m=0, n=0); # s11
```



For more detailed information about plotting see [Plotting](#).

### 1.3.4 Network Operators

#### Linear Operations

Element-wise mathematical operations on the scattering parameter matrices are accessible through overloaded operators. To illustrate their usage, load a couple Networks stored in the `data` module.

```

In [1]: short = rf.data.wr2p2_short
In [2]: delayshort = rf.data.wr2p2_delayshort
In [3]: short - delayshort
Out[3]: 1-Port Network: 'wr2p2,short', 330-500 GHz, 201 pts, z0=[ 50.+0.j]
In [4]: short + delayshort
Out[4]: 1-Port Network: 'wr2p2,short', 330-500 GHz, 201 pts, z0=[ 50.+0.j]
In [5]: short * delayshort
Out[5]: 1-Port Network: 'wr2p2,short', 330-500 GHz, 201 pts, z0=[ 50.+0.j]
In [6]: short / delayshort
Out[6]: 1-Port Network: 'wr2p2,short', 330-500 GHz, 201 pts, z0=[ 50.+0.j]
In [7]: short / delayshort
Out[7]: 1-Port Network: 'wr2p2,short', 330-500 GHz, 201 pts, z0=[ 50.+0.j]

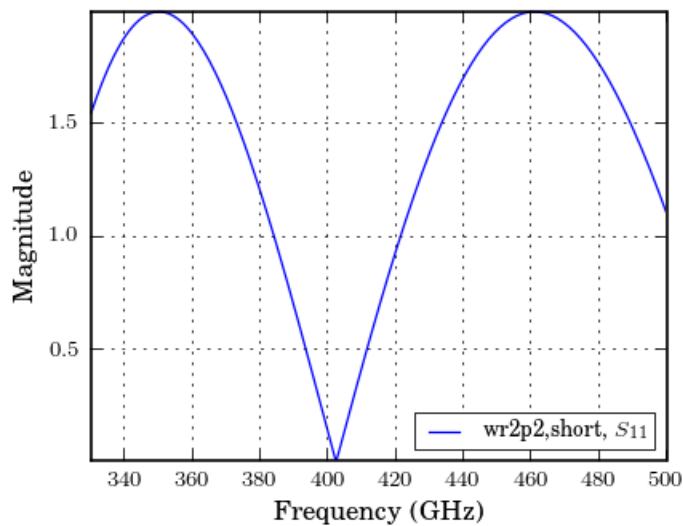
```

All of these operations return `Network` types, so the methods and properties of a `Network` are available on the result. For example, to plot the complex difference between `short` and `delay_short`,

```

In [1]: figure();
In [2]: difference = (short- delayshort)
In [3]: difference.plot_s_mag()

```

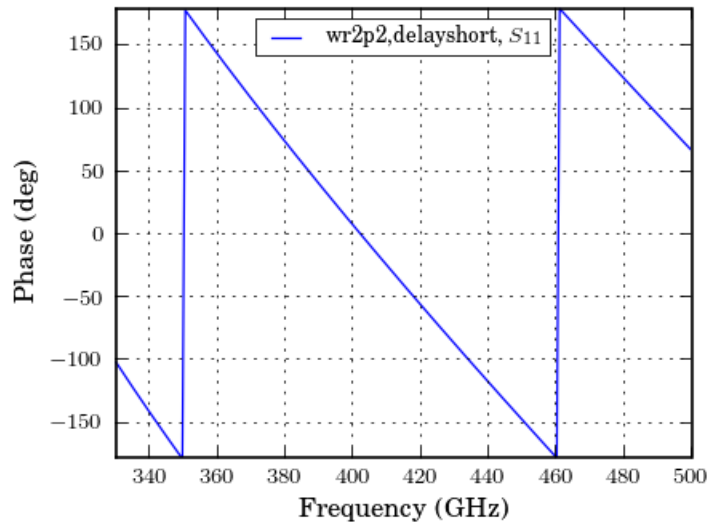


Another common application is calculating the phase difference using the division operator,

```

In [1]: figure();
In [2]: (delayshort/short).plot_s_deg()

```



Linear operators can also be used with scalars or an `numpy.ndarray` that is the same length as the `Network`.

```
In [1]: open = (short*-1)
```

```
In [2]: open.s[:3,...]
```

```
Out[2]:
array([[ 1.-0.j]],
       [[ 1.-0.j]],
       [[ 1.-0.j]])
```

```
In [3]: rando = open * rand(len(open))
```

```
In [4]: rando.s[:3,...]
```

```
Out[4]:
array([[ 0.13594146+0.j]],
       [[ 0.57169729+0.j]],
       [[ 0.77254824+0.j]])
```

Note that if you multiply a `Network` by an `numpy.ndarray` be sure to place the array on right side.

### Cascading and De-embedding

Cascading and de-embedding 2-port `Networks` can also be done through operators. The `cascade()` function can be called through the power operator, `**`. To calculate a new network which is the cascaded connection of the two individual `Networks` `line` and `short`,

```
In [1]: short = rf.data.wr2p2_short
```

```
In [2]: line = rf.data.wr2p2_line
```

```
In [3]: delayshort = line ** short
```

De-embedding can be accomplished by cascading the *inverse* of a network. The inverse of a network is accessed through the property `Network.inv`. To de-embed the `short` from `delay_short`,

```
In [1]: short = line.inv ** delayshort
```

### 1.3.5 Connecting Multi-ports

**skrf** supports the connection of arbitrary ports of N-port networks. It accomplishes this using an algorithm called sub-network growth<sup>1</sup>, available through the function `connect()`. Terminating one port of an ideal 3-way splitter can be done like so,

```
In [1]: tee = rf.Network('../skrf/data/tee.s3p')
```

To connect port *l* of the tee, to port *o* of the delay short,

```
In [1]: terminated_tee = rf.connect(tee, l, delayshort, o)
```

Note that this function takes into account port impedances, and if connecting ports have different port impedances an appropriate impedance mismatch is inserted.

### 1.3.6 Interpolation and Stitching

A common need is to change the number of frequency points of a `Network`. For instance, to use the operators and cascading functions the networks involved must have matching frequencies. If two networks have different frequency information, then an error will be raised,

```
In [1]: line = rf.data.wr2p2_line.copy()
```

```
In [2]: line1 = rf.data.wr2p2_line1.copy()
```

```
In [3]: line1
```

```
Out[3]: 2-Port Network: 'wr2p2_line1', 330-500 GHz, 101 pts, z0=[ 50.+0.j 50.+0.j]
```

```
In [4]: line
```

```
Out[4]: 2-Port Network: 'wr2p2_line', 330-500 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j]
```

```
In [5]: line1+line
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-5-82040f7eab08> in <module>()
----> 1 line1+line
```

```
/home/alex/data/docs/code/path/skrf/network.pyc in __add__(self, other)
    434
    435     if isinstance(other, Network):
--> 436         self.__comparable_for_scalar_operation_test(other)
    437         result.s = self.s + other.s
    438     else:
```

```
/home/alex/data/docs/code/path/skrf/network.pyc in __comparable_for_scalar_operation_test(self, other)
    560     '''
    561     if other.frequency != self.frequency:
--> 562         raise IndexError('Networks must have same frequency. See 'Network.interpolate')
    563
    564     if other.s.shape != self.s.shape:
```

<sup>1</sup> Compton, R.C.; "Perspectives in microwave circuit analysis," Circuits and Systems, 1989., Proceedings of the 32nd Midwest Symposium on , vol., no., pp.716-718 vol.2, 14-16 Aug 1989. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=101955&isnumber=3167>

`IndexError: Networks must have same frequency. See 'Network.interpolate'`

This problem can be solved by interpolating one of Networks, using `Network.resample()`.

```
In [1]: line1
Out[1]: 2-Port Network: 'wr2p2,line1', 330-500 GHz, 101 pts, z0=[ 50.+0.j 50.+0.j]

In [2]: line1.resample(201)

In [3]: line1
Out[3]: 2-Port Network: 'wr2p2,line1', 330-500 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j]

In [4]: line1+line
Out[4]: 2-Port Network: 'wr2p2,line1', 330-500 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j]
```

A related application is the need to combine Networks which cover different frequency ranges. Two Networks can be stitched together using `stitch()`, which concatenates their s-parameter matrices along their frequency axis. To combine a WR-2.2 Network with a WR-1.5 Network,

```
In [1]: from skrf.data import wr2p2_line, wr1p5_line

In [2]: line = rf.stitch(wr2p2_line, wr1p5_line)

In [3]: line
Out[3]: 2-Port Network: 'wr2p2,line', 330-750 GHz, 402 pts, z0=[ 50.+0.j 50.+0.j]
```

### 1.3.7 Reading and Writing

While `skrf` supports reading and writing the touchstone file format, it also provides native IO capabilities for any `skrf` object through the functions `read()` and `write()`. These functions can also be called through the Network methods `Network.read()` and `Network.write()`. The Network constructor (`Network.__init__()`) calls `read()` implicitly if a `skrf` file is passed.

```
In [1]: line = rf.Network('../skrf/data/line.s2p')

In [2]: line.write() # write out Network using native IO
line.ntwk

In [3]: rf.Netwrok('line.ntwk') # read Network using native IO
```

Frequently there is an entire directory of files that need to be analyzed. The function `read_all()` is used to create objects from all files in a directory quickly. Given a directory of `skrf`-readable files, `read_all()` returns a `dict` with keys equal to the filenames, and values equal to objects. To load all `skrf` files in the `skrf/data/` directory which contain the string `'wr2p2'`.

```
In [1]: dict_o_ntwks = rf.read_all('../skrf/data/', contains = 'wr2p2')

In [2]: dict_o_ntwks
Out[2]:
{'wr2p2,delayshort': 1-Port Network: 'wr2p2,delayshort', 330-500 GHz, 201 pts, z0=[ 50.+0.j],
 'wr2p2,line': 2-Port Network: 'wr2p2,line', 330-500 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j],
 'wr2p2,line1': 2-Port Network: 'wr2p2,line1', 330-500 GHz, 101 pts, z0=[ 50.+0.j 50.+0.j],
 'wr2p2,short': 1-Port Network: 'wr2p2,short', 330-500 GHz, 201 pts, z0=[ 50.+0.j]}
```

`read_all()` has a companion function, `write_all()` which takes a dictionary of `skrf` objects, and writes each object to an individual file.

```
In [1]: rf.write_all(dict_o_ntwks, dir = '.')
```

```
In [2]: ls
wr2p2,delayshort.ntwk  wr2p2,line.ntwk          wr2p2,short.ntwk
```

It is also possible to write a dictionary of objects to a single file, by using `write()`,

```
In [1]: rf.write('dict_o_ntwk.p', dict_o_ntwks)
```

```
In [2]: ls
dict_o_ntwk.p
```

A similar function `save_sesh()`, can be used to save all `skrf` objects in the current namespace.

### 1.3.8 Impedance and Admittance Parameters

This tutorial focuses on s-parameters, but other network representations are available as well. Impedance and Admittance Parameters can be accessed through the parameters `Network.z` and `Network.y`, respectively. Scalar components of complex parameters, such as `Network.z_re`, `Network.z_im` and plotting methods like `Network.plot_z_mag()` are available as well.

```
In [1]: ring_slot.z[:3,...]
```

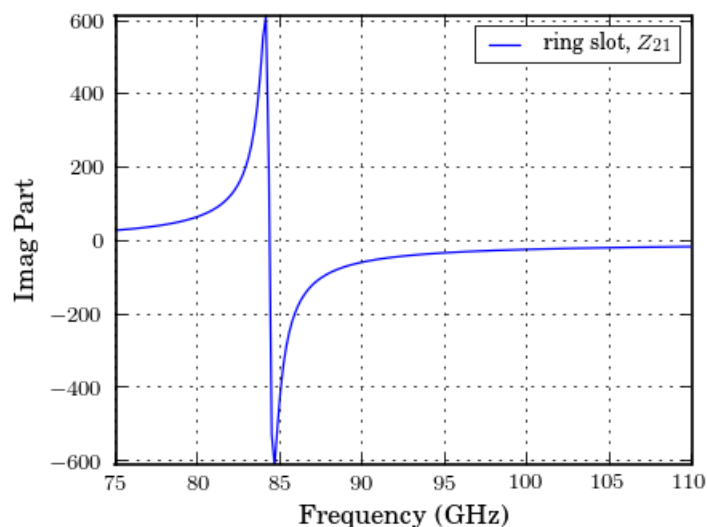
```
Out[1]:
array([[ 0.88442687+28.15350224j,  0.94703504+30.46757222j],
       [ 0.94703504+30.46757222j,  1.04344170+43.45766805j]],

       [[ 0.91624901+28.72415928j,  0.98188607+31.09594438j],
       [ 0.98188607+31.09594438j,  1.08168411+44.17642274j]],

       [[ 0.94991736+29.31694632j,  1.01876516+31.74874257j],
       [ 1.01876516+31.74874257j,  1.12215451+44.92215712j]])
```

```
In [2]: figure();
```

```
In [3]: ring_slot.plot_z_im(m=1,n=0)
```



### 1.3.9 Creating Networks ‘From Scratch’

A `Network` can be created *from scratch* by passing values of relevant properties as keyword arguments to the constructor,

```
In [1]: frequency = rf.Frequency(75,110,101,'ghz')
```

```
In [2]: s = -1*ones(101)
```

```
In [3]: wr10_short = rf.Network(frequency = frequency, s = s, z0 = 50 )
```

For more information creating Networks representing transmission line and lumped components, see the `media` module.

### 1.3.10 Sub-Networks

Frequently, the one-port s-parameters of a multiport network’s are of interest. These can be accessed by the sub-network properties, which return one-port `Network` objects,

```
In [1]: port1_return = line.s11
```

```
In [2]: port1_return
```

```
Out[2]: 1-Port Network: 'line', 75-110 GHz, 201 pts, z0=[ 50.+0.j]
```

### 1.3.11 References

## 1.4 Plotting

### Contents

- Plotting
  - Plotting Methods
  - Complex Plots
    - \* Smith Chart
    - \* Complex Plane
  - Rectangular Plots
    - \* Log-Magnitude
    - \* Phase
    - \* Impedance, Admittance
  - Customizing Plots
  - Saving Plots
  - Misc
    - \* Adding Markers to Lines
    - \* Formating Plots

### 1.4.1 Plotting Methods

Network plotting abilities are implemented as methods of the `Network` class. Some of the plotting functions of network s-parameters are,

- `Network.plot_s_re()`



- `Network.plot_s_im()`
- `Network.plot_s_mag()`
- `Network.plot_s_db()`
- `Network.plot_s_deg()`
- `Network.plot_s_deg_unwrap()`
- `Network.plot_s_rad()`
- `Network.plot_s_rad_unwrap()`
- `Network.plot_s_smith()`
- `Network.plot_s_complex()`

Similar methods exist for Impedance (`Network.z`) and Admittance Parameters (`Network.y`),

- `Network.plot_z_re()`
- `Network.plot_z_im()`
- ...
- `Network.plot_y_re()`
- `Network.plot_y_im()`
- ...

Step-by-step examples of how to create and customize plots are given below.

## 1.4.2 Complex Plots

### Smith Chart

As a first example, load a `Network` from the data module, and plot all four s-parameters on the Smith chart.

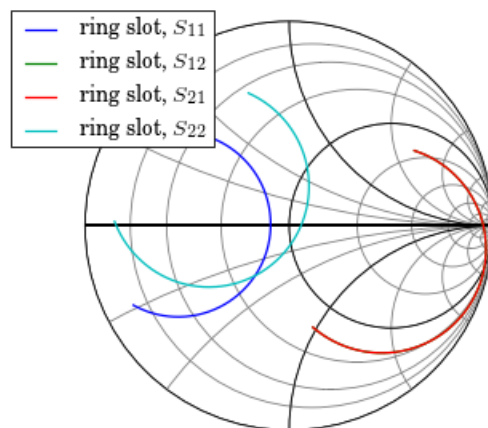
```
In [1]: import skrf as rf
```

```
In [2]: from skrf.data import ring_slot
```

```
In [3]: ring_slot
```

```
Out[3]: 2-Port Network: 'ring slot', 75-110 GHz, 501 pts, z0=[ 50.+0.j 50.+0.j]
```

```
In [4]: ring_slot.plot_s_smith()
```



**Note:** If you don't see any plots after issuing these commands, then you may not have started ipython with the `--pylab` flag. Try `from pylab import *` to import the matplotlib commands and `ion()` to turn on interactive plotting. See [this page](#), for more info on ipython's *pylab* mode.

---

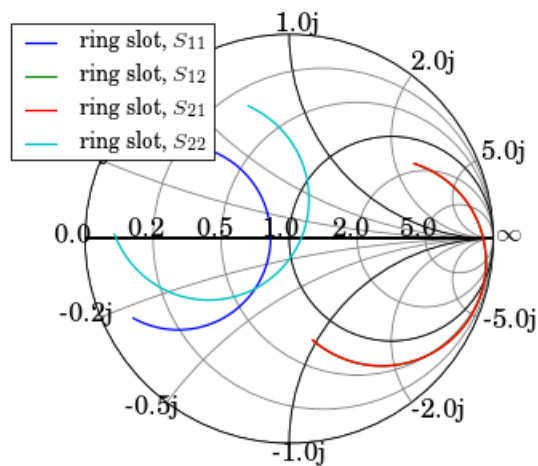
**Note:** Why do my plots look different? See [Formating Plots](#)

---

The smith chart can be drawn with some impedance values labeled through the `draw_labels` argument.

```
In [1]: figure();
```

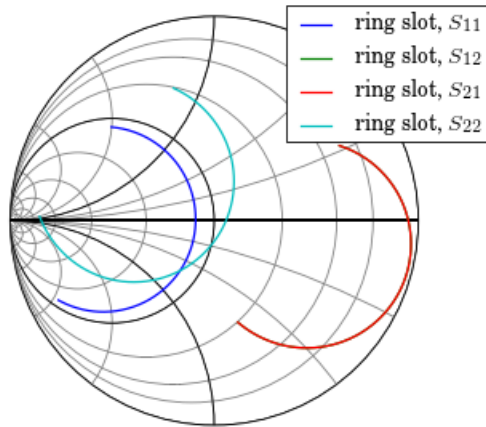
```
In [2]: ring_slot.plot_s_smith(draw_labels=True)
```



Another common option is to draw admittance contours, instead of impedance. This is controlled through the `chart_type` argument.

```
In [1]: figure();
```

```
In [2]: ring_slot.plot_s_smith(chart_type='y')
```



See `smith()` for more info on customizing the Smith Chart.

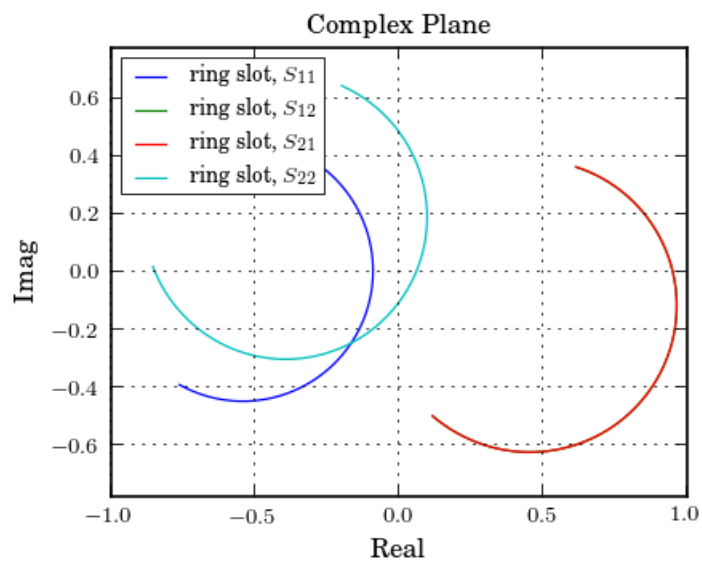
**Note:** If more than one `plot_s_smith()` command is issued on a single figure, you may need to call `draw()` to refresh the chart.

## Complex Plane

Network parameters can also be plotted in the complex plane without a Smith Chart through `Network.plot_s_complex()`.

```
In [1]: figure();
```

```
In [2]: ring_slot.plot_s_complex();
```



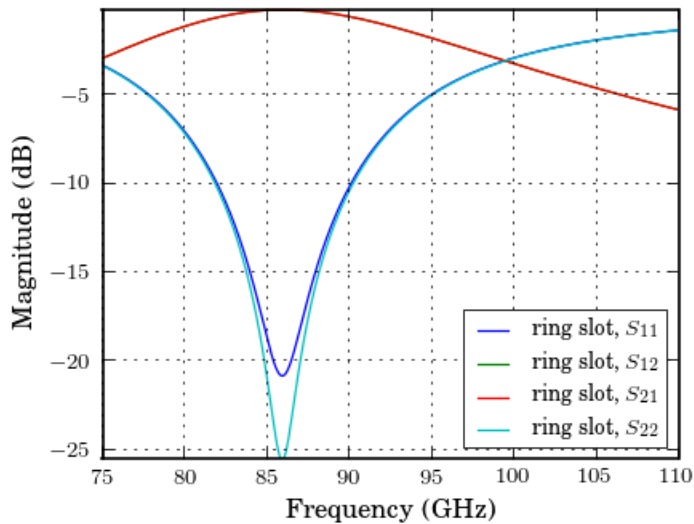
### 1.4.3 Rectangular Plots

#### Log-Magnitude

Scalar components of the complex network parameters can be plotted vs frequency as well. To plot the log-magnitude of the s-parameters vs. frequency,

```
In [1]: figure();
```

```
In [2]: ring_slot.plot_s_db()
```



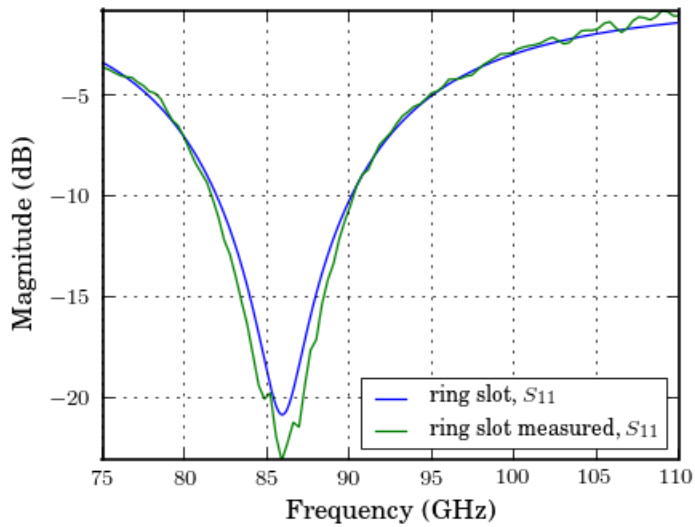
When no arguments are passed to the plotting methods, all parameters are plotted. Single parameters can be plotted by passing indices  $m$  and  $n$  to the plotting commands (indexing start from 0). Comparing the simulated reflection coefficient off the ring slot to a measurement,

```
In [1]: from skrf.data import ring_slot_meas
```

```
In [2]: figure();
```

```
In [3]: ring_slot.plot_s_db(m=0,n=0) # s11
```

```
In [4]: ring_slot_meas.plot_s_db(m=0,n=0) # s11
```



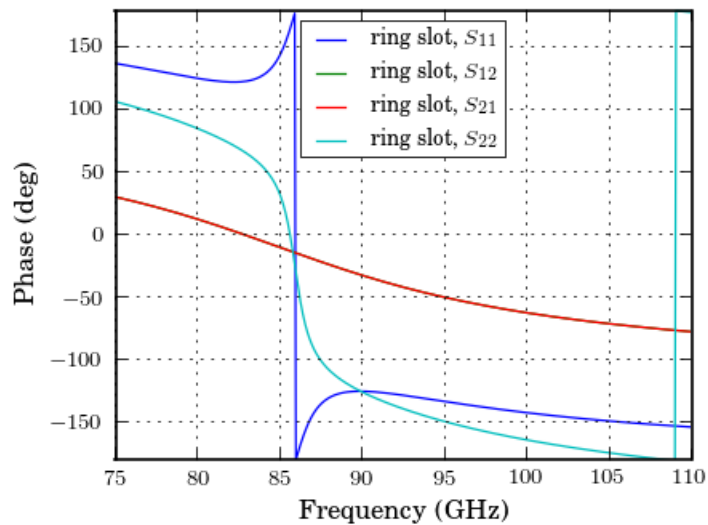
See [Customizing Plots](#) for more information on customization.

## Phase

Plot phase,

```
In [1]: figure();
```

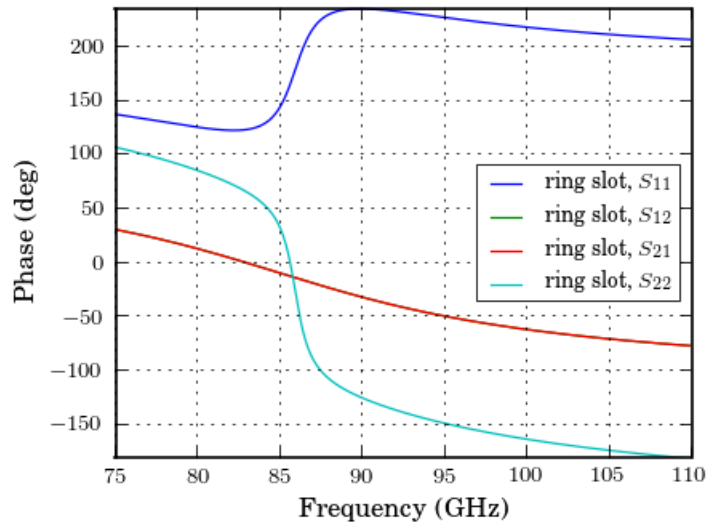
```
In [2]: ring_slot.plot_s_deg();
```



Or unwrapped phase,

```
In [1]: figure();
```

```
In [2]: ring_slot.plot_s_deg_unwrap();
```

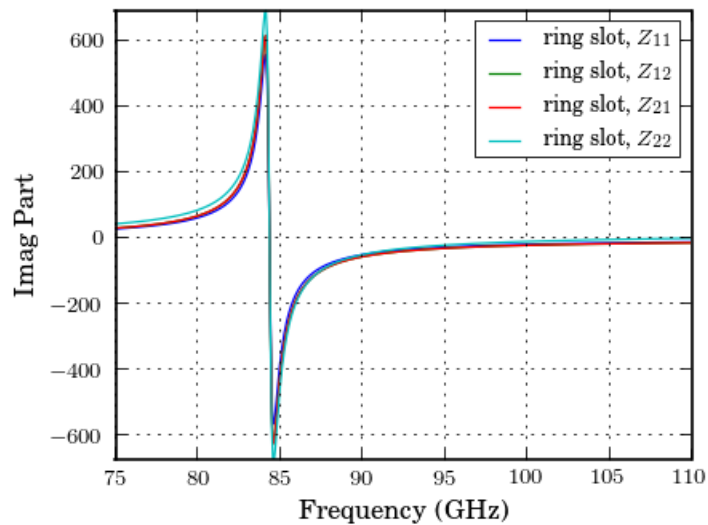


### Impedance, Admittance

The components the Impedance and Admittance parameters can be plotted similarly,

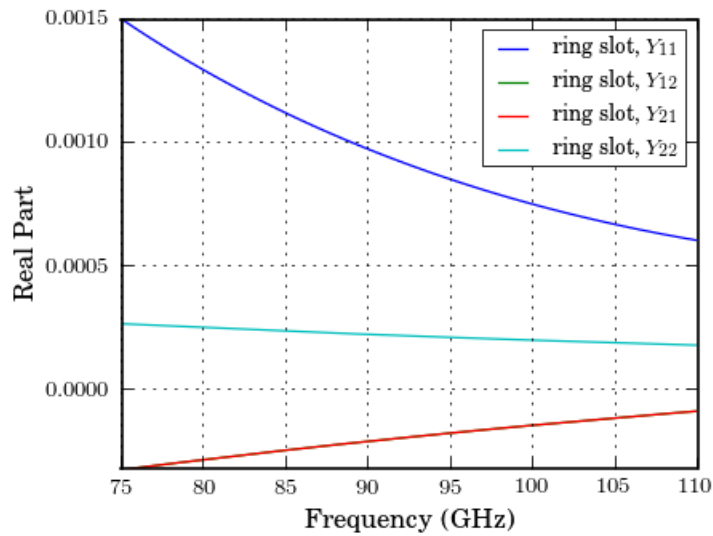
```
In [1]: figure();
```

```
In [2]: ring_slot.plot_z_im()
```



```
In [1]: figure();
```

```
In [2]: ring_slot.plot_y_re()
```



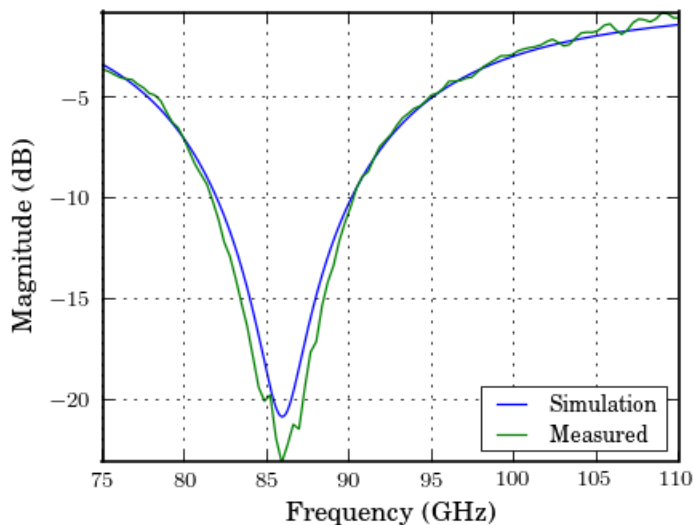
### 1.4.4 Customizing Plots

The legend entries are automatically filled in with the Network's name. The entry can be overridden by passing the `label` argument to the plot method.

```
In [1]: figure();
```

```
In [2]: ring_slot.plot_s_db(m=0,n=0, label = 'Simulation')
```

```
In [3]: ring_slot_meas.plot_s_db(m=0,n=0, label = 'Measured')
```



The frequency unit used on the x-axis is automatically filled in from the Networks `frequency` attribute. To change the label, change the frequency's unit.

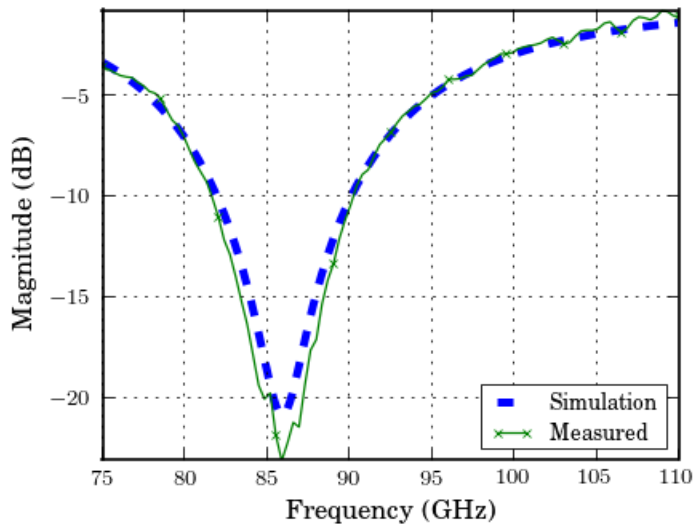
```
In [1]: ring_slot.frequency.unit = 'mhz'
```

Other key word arguments given to the plotting methods are passed through to the matplotlib `plot()` function.

```
In [1]: figure();
```

```
In [2]: ring_slot.plot_s_db(m=0,n=0, linewidth = 3, linestyle = '--', label = 'Simulation')
```

```
In [3]: ring_slot_meas.plot_s_db(m=0,n=0, marker = 'x', markevery = 10, label = 'Measured')
```



All components of the plots can be customized through `matplotlib` functions.

```
In [1]: figure();
```

```
In [2]: ring_slot.plot_s_smith();
```

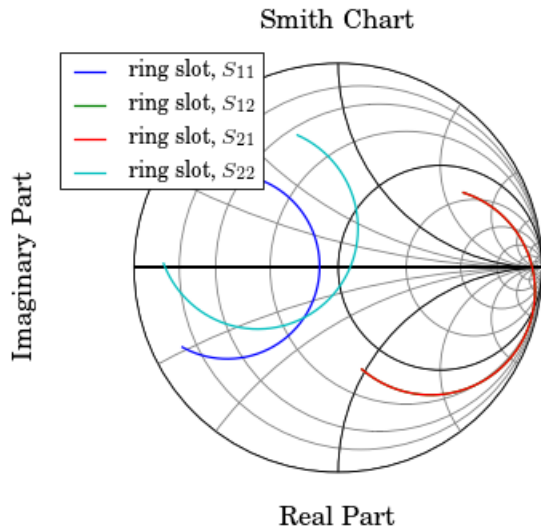
```
In [3]: xlabel('Real Part');
```

```
In [4]: ylabel('Imaginary Part');
```

```
In [5]: title('Smith Chart');
```

```
In [6]: draw();
```





### 1.4.5 Saving Plots

Plots can be saved in various file formats using the GUI provided by the matplotlib. However, skrf provides a convenience function, called `save_all_figs()`, that allows all open figures to be saved to disk in multiple file formats, with filenames pulled from each figure's title:

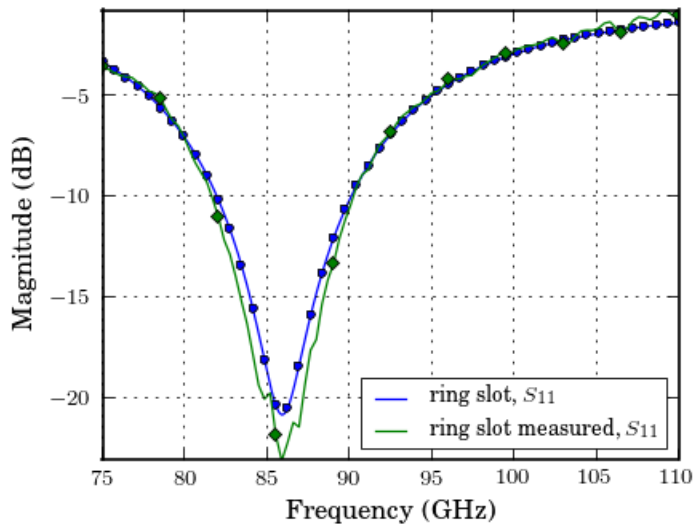
```
>>> rf.save_all_figs('.', format=['eps', 'pdf'])
./WR-10 Ringslot Array Simulated vs Measured.eps
./WR-10 Ringslot Array Simulated vs Measured.pdf
```

### 1.4.6 Misc

#### Adding Markers to Lines

A common need is to make a color plot, interpretable in greyscale print. There is a convenient function, `add_markers_to_lines()`, which adds markers each line in a plots *after* the plot has been made. In this way, adding markers to an already written set of plotting commands is easy.

```
In [1]: figure();
In [2]: ring_slot.plot_s_db(m=0, n=0)
In [3]: ring_slot_meas.plot_s_db(m=0, n=0)
In [4]: rf.add_markers_to_lines()
```



## Formating Plots

It is likely that your plots dont look exactly like the ones in this tutorial. This is because matplotlib supports a vast amount of [customization](#). Formating options can be customized *on-the-fly* by modifying values of the `rcParams` dictionary. Once these are set to your liking they can be saved to your `.matplotlibrc` file.

Here are some relevant parameters which should get your plots looking close to the ones in this tutorial:

```
my_params = {
    'figure.dpi': 120,
    'figure.figsize': [4,3],
    'figure.subplot.left' : 0.15,
    'figure.subplot.right' : 0.9,
    'figure.subplot.bottom' : 0.12,
    'axes.titlesize' : 'medium',
    'axes.labelsize' : 10 ,
    'ytick.labelsize' : 'small',
    'xtick.labelsize' : 'small',
    'legend.fontsize' : 8 #small,
    'legend.loc' : 'best',
    'font.size' : 10.0,
    'font.family' : 'serif',
    'text.usetex' : True, # if you have latex
}
```

```
rcParams.update(my_params)
```

The project [mpltools](#) provides a way to switch between pre-defined *styles*, and contains other useful plotting-related features.

## 1.5 NetworkSet

**Contents**

- NetworkSet
  - Creating a NetworkSet
  - Accesing Network Methods
  - Statistical Properties
  - Plotting Uncertainty Bounds
  - Reading and Writing

The `NetworkSet` object represents an unordered set of networks and provides methods for calculating statistical quantities and displaying uncertainty bounds.

### 1.5.1 Creating a NetworkSet

For this example, assume that numerous measurements of a single network are made. These measurements have been retrieved from a VNA and are in the form of touchstone files. A set of example data can be found in `scikit-rf/skrf/data/`, with naming convention `ro,*.slp`,

```
In [1]: import skrf as rf
```

```
In [2]: ls ../skrf/data/ro*
../skrf/data/ro,1.slp  ../skrf/data/ro,2.slp  ../skrf/data/ro,3.slp
```

The files `ro,1.slp`, `ro,2.slp`, ... are redundant measurements on which we would like to calculate statistics using the `NetworkSet` class.

A `NetworkSet` is created from a list or dict of `Network`'s. So first we need to load all of the touchstone files. This can be done quickly with `read_all()`, which loads all skrf-readable objects in a directory. The argument `contains` is used to load only files which match a given substring.

```
In [1]: rf.read_all('../skrf/data/', contains='ro')
Out[1]:
{'ro,1': 1-Port Network: 'ro,1', 500-750 GHz, 201 pts, z0=[ 50.+0.j],
 'ro,2': 1-Port Network: 'ro,2', 500-750 GHz, 201 pts, z0=[ 50.+0.j],
 'ro,3': 1-Port Network: 'ro,3', 500-750 GHz, 201 pts, z0=[ 50.+0.j]}
```

This can be passed directly to the `NetworkSet` constructor,

```
In [1]: ro_dict = rf.read_all('../skrf/data/', contains='ro')
In [2]: ro_ns = rf.NetworkSet(ro_dict, name='ro set') #name is optional
In [3]: ro_ns
Out[3]: A NetworkSet of length 3
```

A `NetworkSet` can also be constructed from zipfile of touchstones through the class method `NetworkSet.from_zip()`

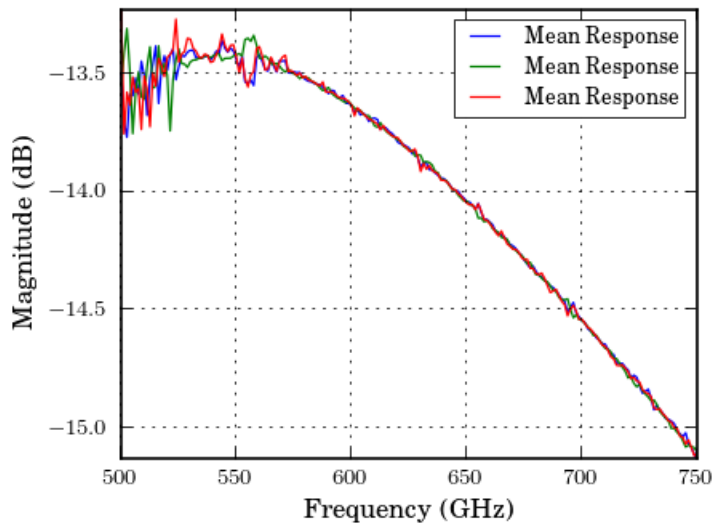
### 1.5.2 Accesing Network Methods

The `Network` elements in a `NetworkSet` can be accessed like the elements of list,

```
In [1]: ro_ns[0]
Out[1]: 1-Port Network: 'ro,1', 500-750 GHz, 201 pts, z0=[ 50.+0.j]
```

Most `Network` methods are also methods of `NetworkSet`. These methods are called on each `Network` element individually. For example to plot the log-magnitude of the s-parameters of each `Network`, (see [Plotting](#) for details on `Network` plotting methods).

```
In [1]: ro_ns.plot_s_db(label='Mean Response')
Out[1]: [None, None, None]
```



### 1.5.3 Statistical Properties

Statistical quantities can be calculated by accessing properties of the `NetworkSet`. For example, to calculate the complex average of the set, access the `mean_s` property

```
In [1]: ro_ns.mean_s
Out[1]: 1-Port Network: 'ro set', 500-750 GHz, 201 pts, z0=[ 50.+0.j]
```

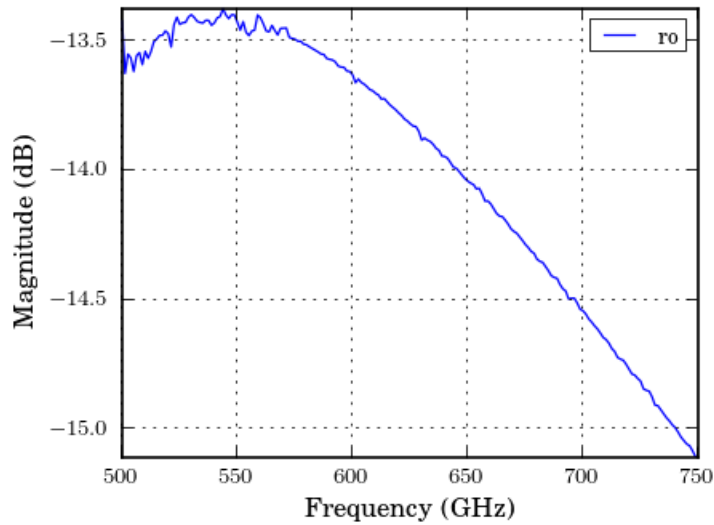
---

**Note:** Because the statistical operator methods are generated upon initialization they are not explicitly documented in this manual.

---

The naming convention of the statistical operator properties are `NetworkSet.function_parameter`, where *function* is the name of the statistical function, and *parameter* is the `Network` parameter to operate on. These methods return a `Network` object, so they can be saved or plotted in the same way as you would with a `Network`. To plot the log-magnitude of the complex mean response

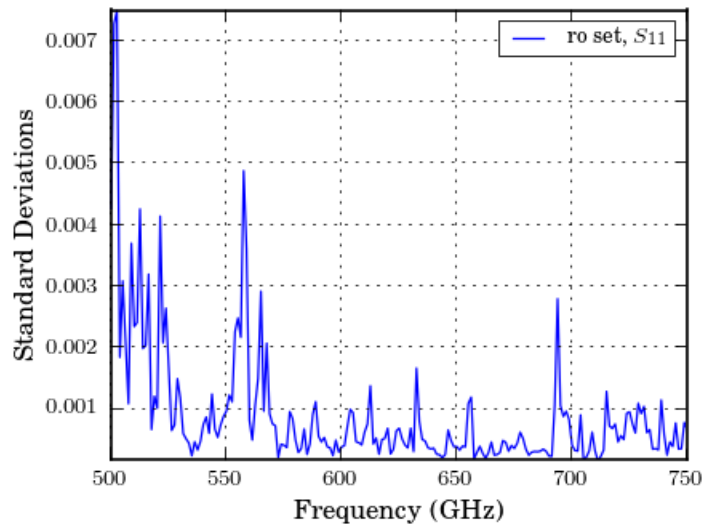
```
In [1]: figure();
In [2]: ro_ns.mean_s.plot_s_db(label='ro')
```



Or to plot the standard deviation of the complex s-parameters,

```
In [1]: figure();
```

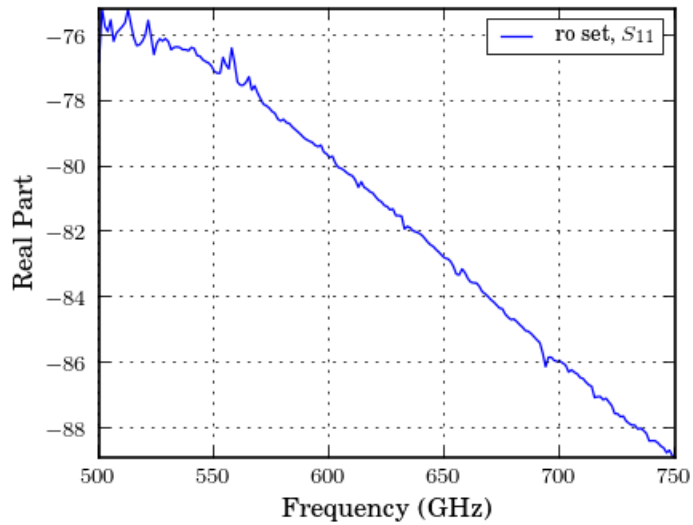
```
In [2]: ro_ns.std_s.plot_s_re(y_label='Standard Deviations')
```



Using these properties it is possible to calculate statistical quantities on the scalar components of the complex network parameters. To calculate the mean of the phase component,

```
In [1]: figure();
```

```
In [2]: ro_ns.mean_s_deg.plot_s_re()
```



### 1.5.4 Plotting Uncertainty Bounds

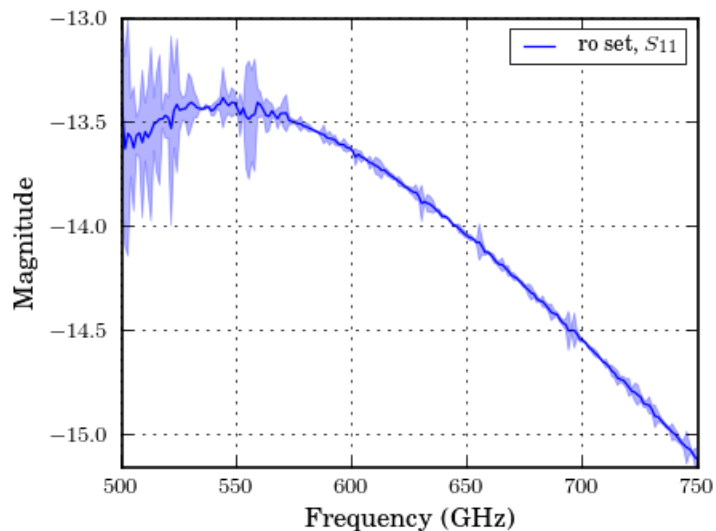
Uncertainty bounds can be plotted through the methods

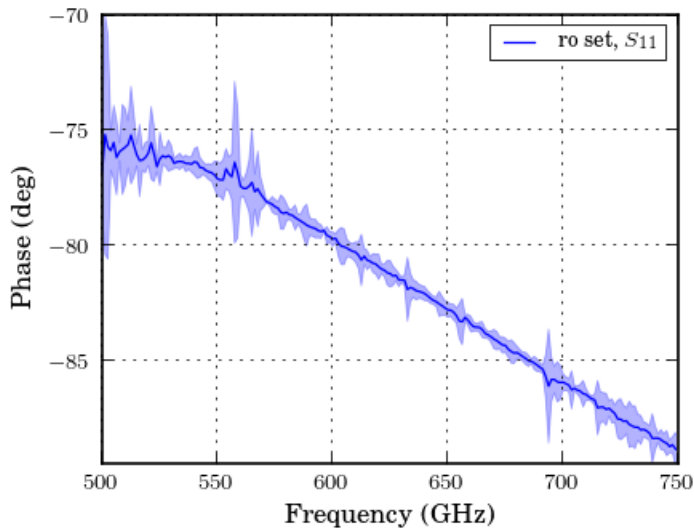
```
In [1]: figure();
```

```
In [2]: ro_ns.plot_uncertainty_bounds_s_db()
```

```
In [3]: figure();
```

```
In [4]: ro_ns.plot_uncertainty_bounds_s_deg()
```





### 1.5.5 Reading and Writing

NetworkSets can be saved to disk using skrf's native IO capabilities. This can be accomplished through the `NetworkSet.write()` method.

```
In [1]: ro_set.write()
```

```
In [2]: ls
ro_set.ns
```

**Note:** Note that if the NetworkSet's name attribute is not assigned, then you must provide a filename to `NetworkSet.write()`.

Alternatively, you can write the Network set by directly calling the `write()` function. In either case, the resultant file can be read back into memory using `read()`.

```
In [1]: ro_ns = rf.read('ro_set.ns')
```

## 1.6 Virtual Instruments

### Contents

- [Virtual Instruments](#)

**Warning:** The `vi` module is not well written or tested at this point.

The `vi` module holds classes for GPIB/VISA instruments that are intricately related to skrf. Most of the classes were created for the sole purpose of retrieving data so that calibration and measurements could be carried out offline with skrf, therefore most other instrument capabilities are neglected.

**Note:** To use the virtual instrument classes you must have `pyvisa` installed, and a working VISA installation.

---

A list of VNA's that have been are partially supported.

- HP8510C
- HP8720
- PNAX
- ZVA40

An example usage of the HP8510C class to retrieve some s-parameter data

```
In [1]: from skrf.vi import vna
```

```
In [2]: my_vna = vna.HP8510C(address =16)
```

```
#if an error is thrown at this point there is most likely a problem with your visa setup
```

```
In [3]: dut_1 = my_vna.s11
```

```
In [4]: dut_2 = my_vna.s21
```

```
In [5]: dut_3 = my_vna.two_port
```

Unfortunately, the syntax is different for every VNA class, so the above example wont directly translate to other VNA's. Re-writing all of the VNA classes to follow the same convention is on the [TODO list](#)

## 1.7 Calibration

### Contents

- Calibration
  - Intro
  - Creating a Calibration
  - Saving and Recalling a Calibration
  - One-Port
  - Concise One-port
  - Two-port
    - \* Switch-terms
  - Example
    - \* Using one-port ideals in two-port Calibration

### 1.7.1 Intro

This tutorial illustrates how to use `skrf` to calibrate data taken from a VNA. The explanation of calibration theory and calibration kit design is beyond the scope of this tutorial. Instead, this tutorial describes how to calibrate a device under test (DUT), assuming you have measured an acceptable set of standards, and have a coresponding set ideal responses.

`skrf`'s default calibration algorithms are generic in that they will work with any set of standards. If you supply more calibration standards than is needed, `skrf` will implement a simple least-squares solution.



## 1.7.2 Creating a Calibration

Calibrations are performed through a `Calibration` class. Creating a `Calibration` object requires at least two pieces of information:

- a list of measured `Network`'s
- a list of ideal `Network`'s

The `Network` elements in each list must all be similar (same #ports, frequency info, etc) and must be aligned to each other, meaning the first element of ideals list must correspond to the first element of measured list.

Optionally, other information can be provided when relevant such as,

- calibration algorithm
- enforce reciprocity of embedding networks
- etc

When this information is not provided `skrf` will determine it through inspection, or use a default value.

## 1.7.3 Saving and Recalling a Calibration

Like other `skrf` objects, `Calibration`'s can be written-to and read-from disk. Writing can be accomplished by using `Calibration.write()`, or `rf.write()`, and reading is done with `rf.read()`.

## 1.7.4 One-Port

This example is written to be instructive, not concise.

```
import skrf as rf

## created necessary data for Calibration class

# a list of Network types, holding 'ideal' responses
my_ideals = [
    rf.Network('ideal/short.slp'),
    rf.Network('ideal/open.slp'),
    rf.Network('ideal/load.slp'),
]

# a list of Network types, holding 'measured' responses
my_measured = [
    rf.Network('measured/short.slp'),
    rf.Network('measured/open.slp'),
    rf.Network('measured/load.slp'),
]

## create a Calibration instance
cal = rf.Calibration(
    ideals = my_ideals,
    measured = my_measured,
)

## run, and apply calibration to a DUT
```

```
# run calibration algorithm
cal.run()

# apply it to a dut
dut = rf.Network('my_dut.slp')
dut_calcd = cal.apply_cal(dut)

# plot results
dut_calcd.plot_s_db()
# save results
dut_calcd.write_touchstone()
```

## 1.7.5 Concise One-port

This example is the same as the first except more concise.

```
import skrf as rf

my_ideals = rf.load_all_touchstones_in_dir('ideals/')
my_measured = rf.load_all_touchstones_in_dir('measured/')

## create a Calibration instance
cal = rf.Calibration(\
    ideals = [my_ideals[k] for k in ['short', 'open', 'load']],
    measured = [my_measured[k] for k in ['short', 'open', 'load']],
)

## what you do with 'cal' may may be similar to above example
```

## 1.7.6 Two-port

Two-port calibration is more involved than one-port. skrf supports two-port calibration using a 8-term error model based on the algorithm described in <sup>2</sup>, by R.A. Speciale.

Like the one-port algorithm, the two-port calibration can handle any number of standards, providing that some fundamental constraints are met. In short, you need three two-port standards; one must be transmissive, and one must provide a known impedance and be reflective.

One draw-back of using the 8-term error model formulation (which is the same formulation used in TRL) is that switch-terms may need to be measured in order to achieve a high quality calibration (this was pointed out to me by Dylan Williams).

### Switch-terms

Originally described by Roger Marks <sup>3</sup>, switch-terms account for the fact that the error networks change slightly depending on which port is being excited. This is due to the internal switch within the VNA.

---

<sup>2</sup> Speciale, R.A.; "A Generalization of the TSD Network-Analyzer Calibration Procedure, Covering n-Port Scattering-Parameter Measurements, Affected by Leakage Errors," Microwave Theory and Techniques, IEEE Transactions on , vol.25, no.12, pp. 1100- 1115, Dec 1977. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1129282&isnumber=25047>

<sup>3</sup> Marks, Roger B.; "Formulations of the Basic Vector Network Analyzer Error Model including Switch-Terms," ARFTG Conference Digest-Fall, 50th , vol.32, no., pp.115-126, Dec. 1997. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4119948&isnumber=4119931>

Switch terms can be measured with a custom measurement configuration on the VNA itself. `skrf` has support for switch terms for the HP8510C class, which you can use or extend to different VNA. Without switch-term measurements, your calibration quality will vary depending on properties of you VNA.

### 1.7.7 Example

Two-port calibration is accomplished in an identical way to one-port, except all the standards are two-port networks. This is even true of reflective standards ( $S_{21}=S_{12}=0$ ). So if you measure reflective standards you must measure two of them simultaneously, and store information in a two-port. For example, connect a short to port-1 and a load to port-2, and save a two-port measurement as 'short,load.s2p' or similar:

```
import skrf as rf

## created necessary data for Calibration class

# a list of Network types, holding 'ideal' responses
my_ideals = [
    rf.Network('ideal/thru.s2p'),
    rf.Network('ideal/line.s2p'),
    rf.Network('ideal/short, short.s2p'),
]

# a list of Network types, holding 'measured' responses
my_measured = [
    rf.Network('measured/thru.s2p'),
    rf.Network('measured/line.s2p'),
    rf.Network('measured/short, short.s2p'),
]

## create a Calibration instance
cal = rf.Calibration(\
    ideals = my_ideals,
    measured = my_measured,
)

## run, and apply calibration to a DUT

# run calibration algorithm
cal.run()

# apply it to a dut
dut = rf.Network('my_dut.s2p')
dut_calcd = cal.apply_cal(dut)

# plot results
dut_calcd.plot_s_db()
# save results
dut_calcd.write_touchstone()
```

### Using one-port ideals in two-port Calibration

Commonly, you have data for ideal data for reflective standards in the form of one-port touchstone files (ie s1p). To use this with `skrf`'s two-port calibration method you need to create a two-port network that is a composite of the two

networks. There is a function in the WorkingBand Class which will do this for you, called `two_port_reflect`:

```
short = rf.Network('ideals/short.slp')
load = rf.Network('ideals/load.slp')
short_load = rf.two_port_reflect(short, load)
```

## Bibliography

## 1.8 Media

### Contents

- Media
  - Introduction
    - \* Media's Supported by skrf
  - Creating Media Objects
    - \* Coplanar Waveguide
    - \* Freespace
    - \* Rectangular Waveguide
  - Working with Media's
  - Network Synthesis
  - Building Cicuits
  - Design Optimization
  - References

### 1.8.1 Introduction

**skrf** supports the microwave network synthesis based on transmission line models. Network creation is accomplished through methods of the Media class, which represents a transmission line object for a given medium. Once constructed, a Media object contains the necessary properties such as propagation constant and characteristic impedance, that are needed to generate microwave circuits.

This tutorial illustrates how created Networks using several different Media objects. The basic usage is,

```
In [1]: import skrf as rf
```

```
In [2]: freq = rf.Frequency(75,110,101,'ghz')
```

```
In [3]: cpw = rf.media.CPW(freq, w=10e-6, s=5e-6, ep_r=10.6)
```

```
In [4]: cpw.line(100*1e-6, name = '100um line')
```

```
Out[4]: 2-Port Network: '100um line', 75-110 GHz, 101 pts, z0=[ 50.06074662+0.j 50.06074662+0.j]
```

More detailed examples illustrating how to create various kinds of Media objects are given below.

**Warning:** The network creation and connection syntax of **skrf** are cumbersome if you need to doing complex circuit design. For a this type of application, you may be interested in using **QUCS** instead. **skrf**'s synthesis capabilities lend themselves more to scripted applications such as [Design Optimization](#) or batch processing.

## Media's Supported by skrf

The base-class, `Media`, is constructed directly from values of propagation constant and characteristic impedance. Specific instances of `Media` objects can be created from relevant physical and electrical properties. Below is a list of mediums types supported by skrf,

- `CPW`
- `RectangularWaveguide`
- `Freespace`
- `DistributedCircuit`
- `Media`

## 1.8.2 Creating Media Objects

Typically, network analysis is done within a given frequency band. When a `Media` object is created, it must be given a `Frequency` object. This prevent having to repeatedly provide frequency information for each new network created.

### Coplanar Waveguide

Here is an example of how to initialize a coplanar waveguide<sup>4</sup> media. The instance has a 10um center conductor, gap of 5um, and substrate with relative permativity of 10.6,

```
In [1]: import skrf as rf
In [2]: freq = rf.Frequency(75,110,101,'ghz')
In [3]: cpw = rf.media.CPW(freq, w=10e-6, s=5e-6, ep_r=10.6)
In [4]: cpw
Out[4]:
Coplanar Waveguide Media. 75-110 GHz. 101 points
W= 1.00e-05m, S= 5.00e-06m
```

See `CPW` for details on that class.

### Freespace

Here is another example, this time constructing a plane-wave in freespace from 10-20GHz

```
In [1]: freq = rf.Frequency(10,20,101,'ghz')
In [2]: fs = rf.media.Freespace(freq)
In [3]: fs
Out[3]: Freespace Media. 10-20 GHz. 101 points
```

See `Freespace` for details.

<sup>4</sup> <http://www.microwaves101.com/encyclopedia/coplanarwaveguide.cfm>

## Rectangular Waveguide

or a WR-10 Rectangular Waveguide

```
In [1]: freq = rf.Frequency(75,110,101,'ghz')
```

```
In [2]: wg = rf.media.RectangularWaveguide(freq, a=100*rf.mil, z0=50) # see note below about z0
```

```
In [3]: wg
```

```
Out [3]:
```

```
Rectangular Waveguide Media. 75-110 GHz. 101 points  
a= 2.54e-03m, b= 1.27e-03m
```

See [RectangularWaveguide](#) for details.

---

**Note:** The `z0` argument in the `Rectangular Waveguide` constructor is used to force a specific port impedance. This is commonly used to match the port impedance to what a VNA stores in a touchstone file. See `media.Media.__init__()` for more information.

---

### 1.8.3 Working with Media's

Once constructed, the pertinent wave quantities of the media such as propagation constant and characteristic impedance can be accessed through the properties `propagation_constant` and `characteristic_impedance`. These properties return complex `numpy.ndarray`'s,

```
In [1]: cpw.propagation_constant[:3]
```

```
Out [1]: array([ 0.+3785.59740815j,  0.+3803.26352939j,  0.+3820.92965062j])
```

```
In [2]: cpw.characteristic_impedance[:3]
```

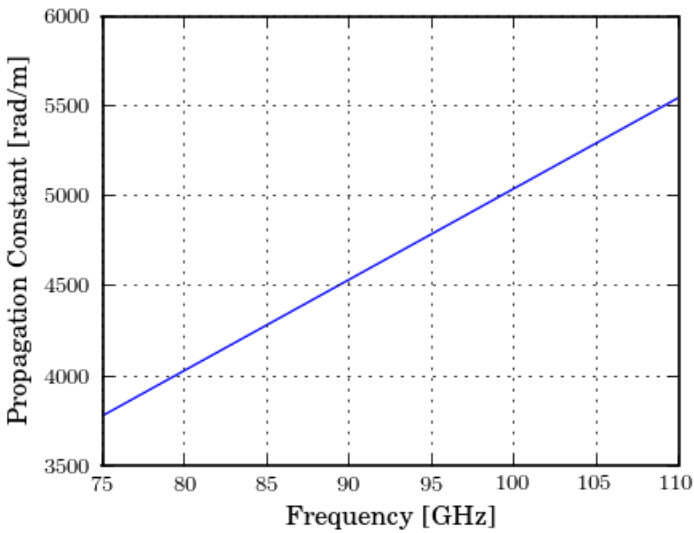
```
Out [2]: array([ 50.06074662+0.j,  50.06074662+0.j,  50.06074662+0.j])
```

As an example, plot the cpw's propagation constant vs frequency.

```
In [1]: plot(cpw.frequency.f_scaled, cpw.propagation_constant.imag);
```

```
In [2]: xlabel('Frequency [GHz]');
```

```
In [3]: ylabel('Propagation Constant [rad/m]');
```



Because the wave quantities are dynamic they change when the attributes of the cpw line change. To illustrate this, plot the propagation constant of the cpw for various values of substrated permativity,

```
In [1]: figure();
```

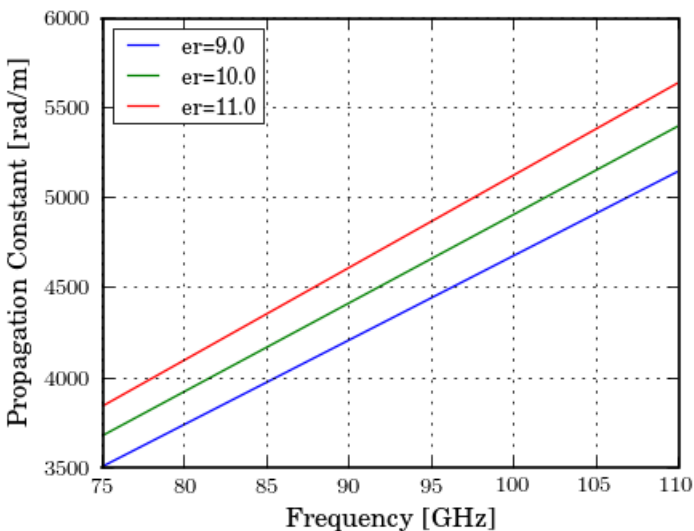
```
In [2]: for ep_r in [9,10,11]:
...:     cpw.ep_r = ep_r
...:     plot(cpw.frequency.f_scaled, cpw.propagation_constant.imag, label='er=%.1f'%ep_r)
...:
```

```
In [3]: xlabel('Frequency [GHz]');
```

```
In [4]: ylabel('Propagation Constant [rad/m]');
```

```
In [5]: legend();
```

```
In [6]: cpw.ep_r = 10.6
```



## 1.8.4 Network Synthesis

Networks are created through methods of a Media object. Here is a brief list of some generic network components skrf supports,

- `match()`
- `short()`
- `open()`
- `load()`
- `line()`
- `thru()`
- `tee()`
- `delay_short()`
- `shunt_delay_open()`

Usage of these methods can is demonstrated below.

To create a 1-port network for a rectangular waveguide short,

```
In [1]: wg.short(name = 'short')
Out[1]: 1-Port Network: 'short', 75-110 GHz, 101 pts, z0=[ 50.+0.j]
```

Or to create a 90° section of cpw line,

```
In [1]: cpw.line(d=90,unit='deg', name='line')
Out[1]: 2-Port Network: 'line', 75-110 GHz, 101 pts, z0=[ 50.06074662+0.j 50.06074662+0.j]
```

---

**Note:** Simple circuits like `short()` and `open()` are ideal short and opens with  $\Gamma = -1$  and  $\Gamma = 1$ , i.e. they dont take into account sophisticated effects of the discontinuities. Effects of discontinuities are implemented as methods specific to a given Media, like `CPW.cpw_short`.

---

## 1.8.5 Building Cicuits

By connecting a series of simple circuits, more complex circuits can be made. To build a the 90° delay short, in the rectangular waveguide media defined above.

```
In [1]: delay_short = wg.line(d=90,unit='deg') ** wg.short()
In [2]: delay_short.name = 'delay short'
In [3]: delay_short
Out[3]: 1-Port Network: 'delay short', 75-110 GHz, 101 pts, z0=[ 50.+0.j]
```

When Networks with more than 2 ports need to be connected together, use `rf.connect()`. To create a two-port network for a shunted delayed open, you can create an ideal 3-way splitter (a 'tee') and conect the delayed open to one of its ports,

```
In [1]: tee = cpw.tee()
In [2]: delay_open = cpw.delay_open(40,'deg')
In [3]: shunt_open = rf.connect(tee,1,delay_open,0)
```



If a specific circuit is created frequently, it may make sense to use a function to create the circuit. This can be done most quickly using lambda

```
In [1]: delay_short = lambda d: wg.line(d, 'deg')**wg.short()
```

```
In [2]: delay_short(90)
```

```
Out[2]: 1-Port Network: '', 75-110 GHz, 101 pts, z0=[ 50.+0.j]
```

This is how many of `skrf`'s network creation methods are made internally.

A more useful example may be to create a function for a shunt-stub tuner, that will work for any media object

```
In [1]: def shunt_stub(media, d0, d1):
...:     return media.line(d0, 'deg')**media.shunt_delay_open(d1, 'deg')
...:
```

```
In [2]: shunt_stub(cpw, 10, 90)
```

```
Out[2]: 2-Port Network: '', 75-110 GHz, 101 pts, z0=[ 50.06074662+0.j 50.06074662+0.j]
```

## 1.8.6 Design Optimization

The abilities of `scipy`'s optimizers can be used to automate network design. In this example, `skrf` is used to automate the single stub design. First, we create a 'cost' function which returns something we want to minimize, such as the reflection coefficient magnitude at band center. Then, one of `scipy`'s minimization algorithms is used to determine the optimal parameters of the stub lengths to minimize this cost.

```
In [1]: from scipy.optimize import fmin
```

```
# the load we are trying to match
```

```
In [2]: load = cpw.load(rf.zl_2_Gamma0(z0=50, z1=100))
```

```
# single stub circuit generator function
```

```
In [3]: def shunt_stub(media, d0, d1):
...:     return media.line(d0, 'deg')**media.shunt_delay_open(d1, 'deg')
...:
```

```
# define the cost function we want to minimize (this uses sloppy namespace)
```

```
In [4]: def cost(d):
...:     return (shunt_stub(cpw, d[0], d[1]) ** load)[100].s_mag.squeeze()
...:
```

```
# initial guess of optimal delay lengths in degrees
```

```
In [5]: d0 = 120, 40 # initial guess
```

```
#determine the optimal delays
```

```
In [6]: d_opt = fmin(cost, (120, 40))
```

```
Optimization terminated successfully.
```

```
Current function value: 0.333333
```

```
Iterations: 65
```

```
Function evaluations: 140
```

```
In [7]: d_opt
```

```
Out[7]: array([ 1.74945025e+02, -9.55405994e-08])
```

## 1.8.7 References

- Development



# EXAMPLES

## 2.1 Visualizing a Single Stub Matching Network

### 2.1.1 Introduction

This example illustrates a way to visualize the design space for a single stub matching network. The matching Network consists of a shunt and series stub arranged as shown below, (image taken from R.M. Weikle's Notes).

A single stub matching network can be designed to produce maximum power transfer to the load,  $Z_L$  at a single frequency. The matching network has two design parameters:

- length of series tline
- length of shunt tline

This script illustrates how to create a plot of reflection coefficient magnitude, vs series and shunt line lengths. The optimal designs are then seen as the minima of a 2D surface.

### 2.1.2 Script

```
import skrf as rf
from pylab import *

# Inputs
wg = rf.wr10 # The Media class
f0 = 90      # Design Frequency in GHz
d_start, d_stop = 0,180 # span of tline lengths [degrees]
n = 21      # number of points
Gamma0 = .5 # the reflection coefficient off the load we are matching

# change wg.frequency so we only simulat at f0
wg.frequency = rf.Frequency(f0,f0,1,'ghz')
# create load network
load = wg.load(.5)
# the vector of possible line-lengths to simulate at
d_range = linspace(d_start,d_stop,n)

def single_stub(wg, d):
    """
    function to return series-shunt stub matching network, given a
```

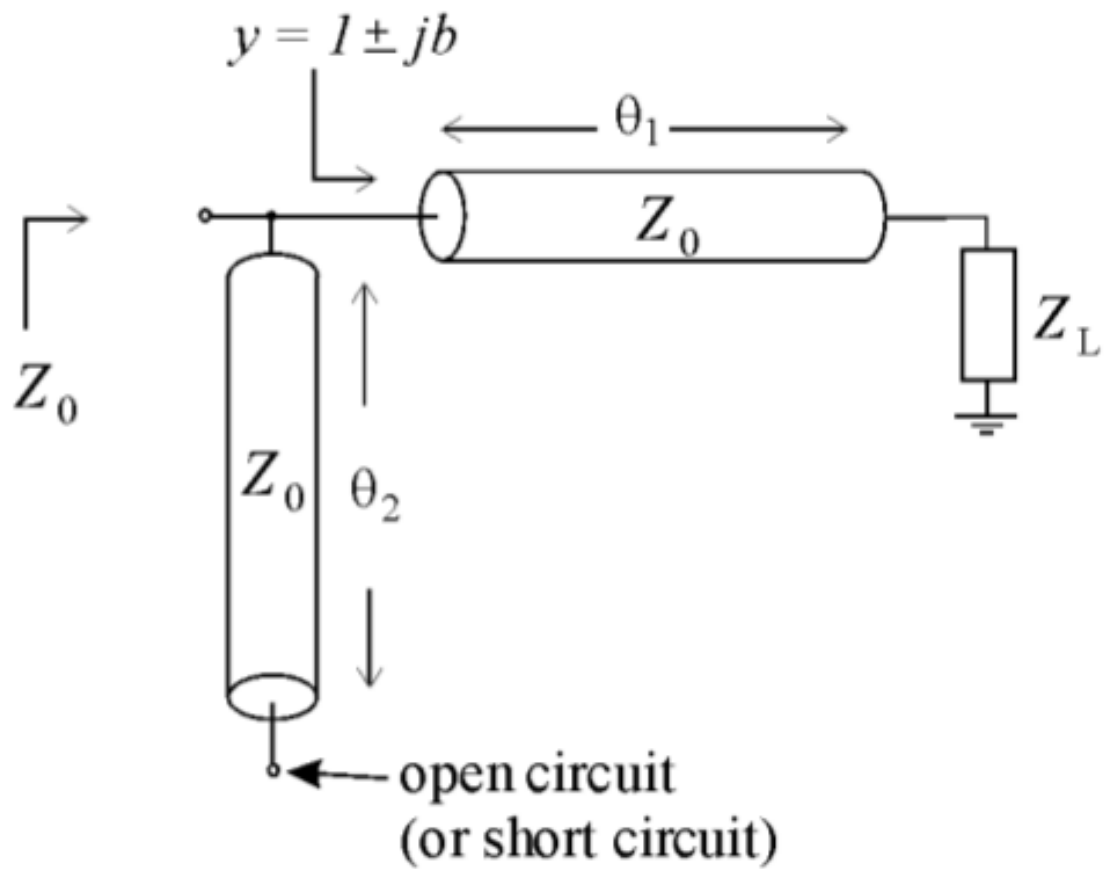


Figure 2.1: Single stub matching Network

```

    WorkingBand and the electrical lengths of the stubs
    '''
    return wg.shunt_delay_open(d[1], 'deg') ** wg.line(d[0], 'deg')

# loop through all line-lengths for series and shunt tlines, and store
# reflection coefficient magnitude in array
output = array([[ (single_stub(wg, [d0,d1])**load).s_mag[0,0,0] \
    for d0 in d_range] for d1 in d_range] )

### Plots

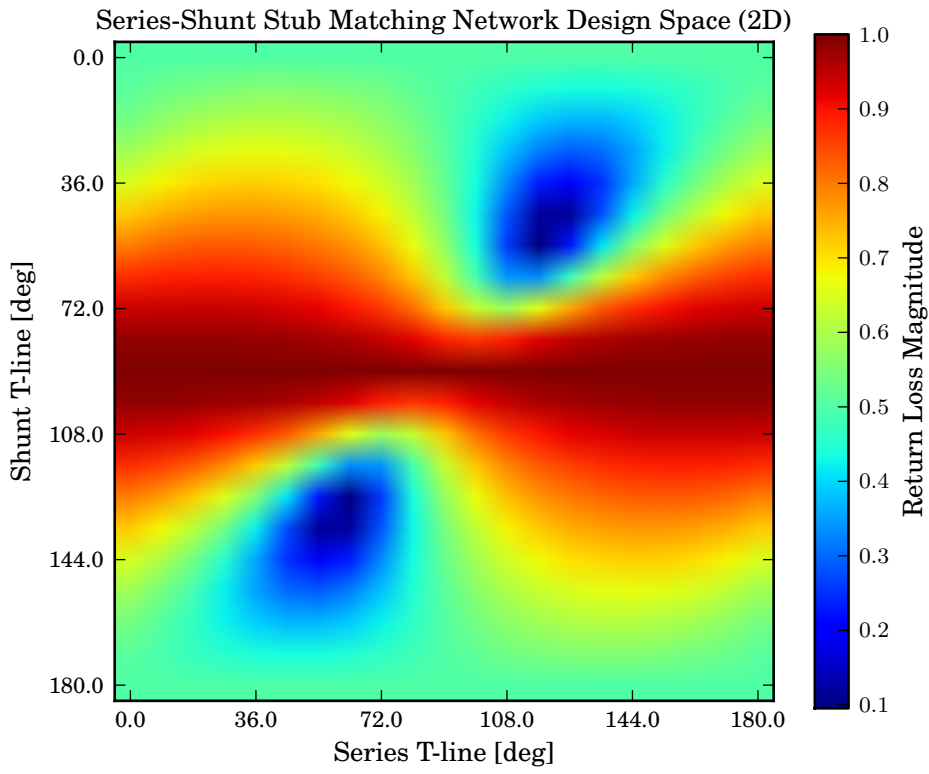
# show the resultant return loss for the parameters space in 2D
figure()
title('Series-Shunt Stub Matching Network Design Space (2D)')
imshow(output)
xlabel('Series T-line [deg]')
ylabel('Shunt T-line [deg]')
xticks(range(0,n+1,n/5),d_range[0::n/5])
yticks(range(0,n+1,n/5),d_range[0::n/5])
cbar = colorbar()
cbar.set_label('Return Loss Magnitude')
grid(False)

# show the resultant return loss for the parameters space in 3D
from mpl_toolkits.mplot3d import Axes3D

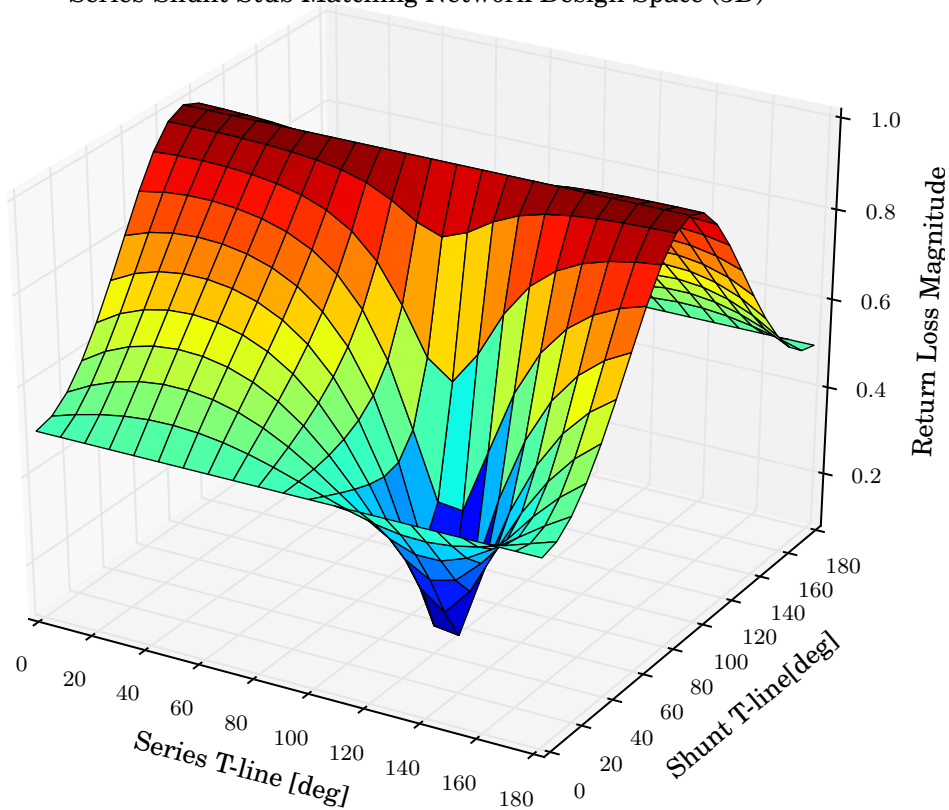
fig=figure()
ax = Axes3D(fig)
x,y = meshgrid(d_range, d_range)
ax.plot_surface(x,y,output, rstride=1, cstride=1,cmap=cm.jet)
ax.set_xlabel('Series T-line [deg]')
ax.set_ylabel('Shunt T-line [deg]')
ax.set_zlabel('Return Loss Magnitude')
ax.set_title(r'Series-Shunt Stub Matching Network Design Space (3D)')

show()

```



Series-Shunt Stub Matching Network Design Space (3D)



## 2.2 One-Port Calibration

### 2.2.1 Instructive

This example is written to be instructive, not concise.:

```
import skrf as rf

## created necessary data for Calibration class

# a list of Network types, holding 'ideal' responses
my_ideals = [
    rf.Network('ideal/short.slp'),
    rf.Network('ideal/open.slp'),
    rf.Network('ideal/load.slp'),
]

# a list of Network types, holding 'measured' responses
my_measured = [
    rf.Network('measured/short.slp'),
    rf.Network('measured/open.slp'),
    rf.Network('measured/load.slp'),
]
```

```
## create a Calibration instance
cal = rf.Calibration(\
    ideals = my_ideals,
    measured = my_measured,
)

## run, and apply calibration to a DUT

# run calibration algorithm
cal.run()

# apply it to a dut
dut = rf.Network('my_dut.slp')
dut_caled = cal.apply_cal(dut)

# plot results
dut_caled.plot_s_db()
# save results
dut_caled.write_touchstone()
```

## 2.2.2 Concise

This example is meant to be the same as the first except more concise:

```
import skrf as rf

my_ideals = rf.load_all_touchstones_in_dir('ideals/')
my_measured = rf.load_all_touchstones_in_dir('measured/')

## create a Calibration instance
cal = rf.Calibration(\
    ideals = [my_ideals[k] for k in ['short', 'open', 'load']],
    measured = [my_measured[k] for k in ['short', 'open', 'load']],
)

## what you do with 'cal' may be similar to above example
```



# REFERENCE

## 3.1 frequency (`skrf.frequency`)

Provides a frequency object and related functions.

Most of the functionality is provided as methods and properties of the `Frequency` Class.

### 3.1.1 Frequency Class

---

`Frequency([start, stop, npoints, unit, ...])` A frequency band.

---

#### `skrf.frequency.Frequency`

**class** `skrf.frequency.Frequency` (*start=0, stop=0, npoints=0, unit='ghz', sweep\_type='lin'*)  
A frequency band.

The frequency object provides a convenient way to work with and access a frequency band. It contains a frequency vector as well as a frequency unit. This allows a frequency vector in a given unit to be available (`f_scaled`), as well as an absolute frequency axis in 'Hz' (`f`).

A Frequency object can be created from either (`start, stop, npoints`) using the default constructor, `__init__()`. Or, it can be created from an arbitrary frequency vector by using the class method `from_f()`.

Internally, the frequency information is stored in the `f` property combined with the `unit` property. All other properties, `start stop`, etc are generated from these.

#### Attributes

<code>center</code>	Center frequency.
<code>f</code>	Frequency vector in Hz
<code>f_scaled</code>	Frequency vector in units of <code>unit</code>
<code>multiplier</code>	Multiplier for forming axis
<code>multiplier_dict</code>	
<code>npoints</code>	starting frequency in Hz
<code>span</code>	the frequency span
<code>start</code>	starting frequency in Hz
<code>step</code>	the inter-frequency step size
Continued on next page	

Table 3.2 – continued from previous page

<code>stop</code>	starting frequency in Hz
<code>unit</code>	Unit of this frequency band.
<code>unit_dict</code>	
<code>w</code>	Frequency vector in radians/s

**skrf.frequency.Frequency.center**Frequency.**center**

Center frequency.

**Returns center** : numberthe exact center frequency in units of `unit`**skrf.frequency.Frequency.f**Frequency.**f**

Frequency vector in Hz

**Returns f** : `numpy.ndarray`

The frequency vector in Hz

**See Also:****f\_scaled** frequency vector in units of `unit`**w** angular frequency vector in rad/s**skrf.frequency.Frequency.f\_scaled**Frequency.**f\_scaled**Frequency vector in units of `unit`**Returns f\_scaled** : `numpy.ndarray`A frequency vector in units of `unit`**See Also:****f** frequency vector in Hz**w** frequency vector in rad/s**skrf.frequency.Frequency.multiplier**Frequency.**multiplier**

Multiplier for formatting axis

This accesses the internal dictionary `multiplier_dict` using the value of `unit`**Returns multiplier** : number

multiplier for this Frequencies unit

**skrf.frequency.Frequency.multiplier\_dict**

```
Frequency.multiplier_dict = {'hz': 1, 'khz': 1000.0, 'mhz': 1000000.0, 'thz': 1000000000000.0, 'ghz': 1000000000.0}
```

**skrf.frequency.Frequency.npoints**

```
Frequency.npoints
    starting frequency in Hz
```

**skrf.frequency.Frequency.span**

```
Frequency.span
    the frequency span
```

**skrf.frequency.Frequency.start**

```
Frequency.start
    starting frequency in Hz
```

**skrf.frequency.Frequency.step**

```
Frequency.step
    the inter-frequency step size
```

**skrf.frequency.Frequency.stop**

```
Frequency.stop
    starting frequency in Hz
```

**skrf.frequency.Frequency.unit**

```
Frequency.unit
    Unit of this frequency band.
```

**Possible strings for this attribute are:** 'hz', 'khz', 'mhz', 'ghz', 'thz'

Setting this attribute is not case sensitive.

**Returns unit:** string

lower-case string representing the frequency units

**skrf.frequency.Frequency.unit\_dict**

```
Frequency.unit_dict = {'hz': 'Hz', 'khz': 'KHz', 'mhz': 'MHz', 'thz': 'THz', 'ghz': 'GHz'}
```

**skrf.frequency.Frequency.w**`Frequency.w`

Frequency vector in radians/s

The frequency vector in rad/s

**Returns** `w`: `numpy.ndarray`

The frequency vector in rad/s

**See Also:**`f_scaled` frequency vector in units of `unit``f` frequency vector in Hz**Methods**

<code>__init__</code>	Frequency initializer.
<code>copy</code>	returns a new copy of this frequency
<code>from_f</code>	Alternative constructor of a Frequency object from a frequency
<code>labelXAxis</code>	Label the x-axis of a plot.

**skrf.frequency.Frequency.\_\_init\_\_**`Frequency.__init__` (`start=0`, `stop=0`, `npoints=0`, `unit='ghz'`, `sweep_type='lin'`)

Frequency initializer.

Creates a Frequency object from start/stop/npoints and a unit. Alternatively, the class method `from_f()` can be used to create a Frequency object from a frequency vector instead.

**Parameters** `start`: numberstart frequency in units of `unit`**stop**: numberstop frequency in units of `unit`**npoints**: int

number of points in the band.

**unit**: ['hz', 'khz', 'mhz', 'ghz']frequency unit of the band. This is used to create the attribute `f_scaled`. It is also used by the `Network` class for plots vs. frequency.**See Also:**`from_f` constructs a Frequency object from a frequency vector instead of start/stop/npoints.**Notes**

The attribute `unit` sets the property `freqMultiplier`, which is used to scale the frequency when `f_scaled` is referenced.

## Examples

```
>>> wrlp5band = Frequency(500,750,401, 'ghz')
```

### skrf.frequency.Frequency.copy

`Frequency.copy()`

returns a new copy of this frequency

### skrf.frequency.Frequency.from\_f

**classmethod** `Frequency.from_f(f, *args, **kwargs)`

Alternative constructor of a Frequency object from a frequency vector, the unit of which is set by kwarg 'unit'

**Parameters** `f`: array-like

frequency vector

**\*args, \*\*kwargs**: arguments, keyword arguments

passed on to `__init__()`.

**Returns** `myfrequency`: `Frequency` object

the Frequency object

## Examples

```
>>> f = np.linspace(75,100,101)
>>> rf.Frequency.from_f(f, unit='ghz')
```

### skrf.frequency.Frequency.labelXAxis

`Frequency.labelXAxis(ax=None)`

Label the x-axis of a plot.

Sets the labels of a plot using `matplotlib.pyplot.xlabel()` with string containing the frequency unit.

**Parameters** `ax`: `matplotlib.Axes`, optional

Axes on which to label the plot, defaults what is returned by `matplotlib.pyplot.gca()`

## 3.2 network (skrf.network)

Provides a n-port network class and associated functions.

Most of the functionality in this module is provided as methods and properties of the `Network Class`.

### 3.2.1 Network Class

`Network([file, name, comments])` A n-port electrical network [\[#\]\\_](#).

## skrf.network.Network

**class** `skrf.network.Network` (*file=None, name=None, comments=None, \*\*kwargs*)  
 A n-port electrical network <sup>1</sup>.

For instructions on how to create Network see `__init__()`.

**A n-port network may be defined by three quantities,**

- network parameter matrix (s, z, or y-matrix)
- port characteristic impedance matrix
- frequency information

The `Network` class stores these data structures internally in the form of complex `numpy.ndarray`'s. These arrays are not interfaced directly but instead through the use of the properties:

Property	Meaning
<code>s</code>	scattering parameter matrix
<code>z0</code>	characteristic impedance matrix
<code>f</code>	frequency vector

Although these docs focus on s-parameters, other equivalent network representations such as `z` and `y` are available. Scalar projections of the complex network parameters are accesable through properties as well. These also return `numpy.ndarray`'s.

Property	Meaning
<code>s_re</code>	real part of the s-matrix
<code>s_im</code>	imaginary part of the s-matrix
<code>s_mag</code>	magnitude of the s-matrix
<code>s_db</code>	magnitude in log scale of the s-matrix
<code>s_deg</code>	phase of the s-matrix in degrees

The following operations act on the networks s-matrix.

Operator	Function
<code>+</code>	element-wise addition of the s-matrix
<code>-</code>	element-wise difference of the s-matrix
<code>*</code>	element-wise multiplication of the s-matrix
<code>/</code>	element-wise division of the s-matrix
<code>**</code>	cascading (only for 2-ports)
<code>//</code>	de-embedding (for 2-ports, see <code>inv</code> )

Different components of the `Network` can be visualized through various plotting methods. These methods can be used to plot individual elements of the s-matrix or all at once. For more info about plotting see the [Plotting tutorial](#).

---

<sup>1</sup> [http://en.wikipedia.org/wiki/Two-port\\_network](http://en.wikipedia.org/wiki/Two-port_network)

Method	Meaning
<code>plot_s_smith()</code>	plot complex s-parameters on smith chart
<code>plot_s_re()</code>	plot real part of s-parameters vs frequency
<code>plot_s_im()</code>	plot imaginary part of s-parameters vs frequency
<code>plot_s_mag()</code>	plot magnitude of s-parameters vs frequency
<code>plot_s_db()</code>	plot magnitude (in dB) of s-parameters vs frequency
<code>plot_s_deg()</code>	plot phase of s-parameters (in degrees) vs frequency
<code>plot_s_deg_unwrap()</code>	plot phase of s-parameters (in unwrapped degrees) vs frequency

`Network` objects can be created from a touchstone or pickle file (see `__init__()`), by a `Media` object, or manually by assigning the network properties directly. `Network` objects can be saved to disk in the form of touchstone files with the `write_touchstone()` method.

An exhaustive list of `Network` Methods and Properties (Attributes) are given below

## References

### Attributes

<code>a</code>	Active scattering parameter matrix.
<code>a_arcl</code>	The arcl component of the a-matrix ..
<code>a_arcl_unwrap</code>	The arcl_unwrap component of the a-matrix ..
<code>a_db</code>	The db component of the a-matrix ..
<code>a_deg</code>	The deg component of the a-matrix ..
<code>a_deg_unwrap</code>	The deg_unwrap component of the a-matrix ..
<code>a_im</code>	The im component of the a-matrix ..
<code>a_mag</code>	The mag component of the a-matrix ..
<code>a_rad</code>	The rad component of the a-matrix ..
<code>a_rad_unwrap</code>	The rad_unwrap component of the a-matrix ..
<code>a_re</code>	The re component of the a-matrix ..
<code>f</code>	the frequency vector for the network, in Hz.
<code>frequency</code>	frequency information for the network.
<code>inv</code>	a <code>Network</code> object with ‘inverse’ s-parameters.
<code>nports</code>	the number of ports the network has.
<code>number_of_ports</code>	the number of ports the network has.
<code>passivity</code>	passivity metric for a multi-port network.
<code>s</code>	Scattering parameter matrix.
<code>s11</code>	one-port sub-network.
<code>s12</code>	one-port sub-network.
<code>s21</code>	one-port sub-network.
<code>s22</code>	one-port sub-network.
<code>s_arcl</code>	The arcl component of the s-matrix ..
<code>s_arcl_unwrap</code>	The arcl_unwrap component of the s-matrix ..
<code>s_db</code>	The db component of the s-matrix ..
<code>s_deg</code>	The deg component of the s-matrix ..
<code>s_deg_unwrap</code>	The deg_unwrap component of the s-matrix ..
<code>s_im</code>	The im component of the s-matrix ..
<code>s_mag</code>	The mag component of the s-matrix ..
<code>s_rad</code>	The rad component of the s-matrix ..
<code>s_rad_unwrap</code>	The rad_unwrap component of the s-matrix ..
<code>s_re</code>	The re component of the s-matrix ..

Continued on next page

Table 3.5 – continued from previous page

<code>t</code>	Scattering transfer parameters
<code>y</code>	Admittance parameter matrix.
<code>y_arcl</code>	The arcl component of the y-matrix ..
<code>y_arcl_unwrap</code>	The arcl_unwrap component of the y-matrix ..
<code>y_db</code>	The db component of the y-matrix ..
<code>y_deg</code>	The deg component of the y-matrix ..
<code>y_deg_unwrap</code>	The deg_unwrap component of the y-matrix ..
<code>y_im</code>	The im component of the y-matrix ..
<code>y_mag</code>	The mag component of the y-matrix ..
<code>y_rad</code>	The rad component of the y-matrix ..
<code>y_rad_unwrap</code>	The rad_unwrap component of the y-matrix ..
<code>y_re</code>	The re component of the y-matrix ..
<code>z</code>	Impedance parameter matrix.
<code>z0</code>	Characteristic impedance[s] of the network ports.
<code>z_arcl</code>	The arcl component of the z-matrix ..
<code>z_arcl_unwrap</code>	The arcl_unwrap component of the z-matrix ..
<code>z_db</code>	The db component of the z-matrix ..
<code>z_deg</code>	The deg component of the z-matrix ..
<code>z_deg_unwrap</code>	The deg_unwrap component of the z-matrix ..
<code>z_im</code>	The im component of the z-matrix ..
<code>z_mag</code>	The mag component of the z-matrix ..
<code>z_rad</code>	The rad component of the z-matrix ..
<code>z_rad_unwrap</code>	The rad_unwrap component of the z-matrix ..
<code>z_re</code>	The re component of the z-matrix ..

**skrf.network.Network.a**`Network.a`

Active scattering parameter matrix.

Active scattering parameters are simply inverted s-parameters, defined as  $a = 1/s$ . Useful in analysis of active networks. The a-matrix is a 3-dimensional `numpy.ndarray` which has shape  $fxnxn$ , where  $f$  is frequency axis and  $n$  is number of ports. Note that indexing starts at 0, so  $a[1]$  can be accessed by taking the slice `a[:,0,0]`.

**Returns** `a` : complex `numpy.ndarray` of shape  $fxnxn$

the active scattering parameter matrix.

**See Also:**

`s`, `y`, `z`, `t`, `a`

**skrf.network.Network.a\_arcl**`Network.a_arcl`

The arcl component of the a-matrix

**See Also:**

`a`



**skrf.network.Network.a\_arcl\_unwrap**`Network.a_arcl_unwrap`

The arcl\_unwrap component of the a-matrix

**See Also:**

[a](#)

**skrf.network.Network.a\_db**`Network.a_db`

The db component of the a-matrix

**See Also:**

[a](#)

**skrf.network.Network.a\_deg**`Network.a_deg`

The deg component of the a-matrix

**See Also:**

[a](#)

**skrf.network.Network.a\_deg\_unwrap**`Network.a_deg_unwrap`

The deg\_unwrap component of the a-matrix

**See Also:**

[a](#)

**skrf.network.Network.a\_im**`Network.a_im`

The im component of the a-matrix

**See Also:**

[a](#)

**skrf.network.Network.a\_mag**`Network.a_mag`

The mag component of the a-matrix

**See Also:**

[a](#)

### `skrf.network.Network.a_rad`

#### `Network.a_rad`

The rad component of the a-matrix

#### See Also:

`a`

### `skrf.network.Network.a_rad_unwrap`

#### `Network.a_rad_unwrap`

The rad\_unwrap component of the a-matrix

#### See Also:

`a`

### `skrf.network.Network.a_re`

#### `Network.a_re`

The re component of the a-matrix

#### See Also:

`a`

### `skrf.network.Network.f`

#### `Network.f`

the frequency vector for the network, in Hz.

**Returns** `f`: `numpy.ndarray`

frequency vector in Hz

#### See Also:

`frequency` frequency property that holds all frequency information

### `skrf.network.Network.frequency`

#### `Network.frequency`

frequency information for the network.

This property is a `Frequency` object. It holds the frequency vector, as well frequency unit, and provides other properties related to frequency information, such as start, stop, etc.

**Returns** `frequency`: `Frequency` object

frequency information for the network.

#### See Also:

`f` property holding frequency vector in Hz

`change_frequency` updates frequency property, and interpolates s-parameters if needed

`interpolate` interpolate function based on new frequency info

### `skrf.network.Network.inv`

`Network.inv`

a `Network` object with ‘inverse’ s-parameters.

This is used for de-embedding. It is defined so that the inverse of a `Network` cascaded with itself is unity.

**Returns** `inv` : a `Network` object

a `Network` object with ‘inverse’ s-parameters.

**See Also:**

`inv` function which implements the inverse s-matrix

### `skrf.network.Network.nports`

`Network.nports`

the number of ports the network has.

**Returns** `number_of_ports` : number

the number of ports the network has.

### `skrf.network.Network.number_of_ports`

`Network.number_of_ports`

the number of ports the network has.

**Returns** `number_of_ports` : number

the number of ports the network has.

### `skrf.network.Network.passivity`

`Network.passivity`

passivity metric for a multi-port network.

This returns a matrix whose diagonals are equal to the total power received at all ports, normalized to the power at a single excitation port.

mathematically, this is a test for unitary-ness of the s-parameter matrix <sup>2</sup>.

for two port this is

$$(|S_{11}|^2 + |S_{21}|^2, |S_{22}|^2 + |S_{12}|^2)$$

in general it is

$$S^H \cdot S$$

where  $H$  is conjugate transpose of  $S$ , and  $\cdot$  is dot product.

**Returns** `passivity` : `numpy.ndarray` of shape  $fxn \times n$

<sup>2</sup> [http://en.wikipedia.org/wiki/Scattering\\_parameters#Lossless\\_networks](http://en.wikipedia.org/wiki/Scattering_parameters#Lossless_networks)

## References

### skrf.network.Network.s

Network.**s**

Scattering parameter matrix.

The `s`-matrix[#]\_ is a 3-dimensional `numpy.ndarray` which has shape  $fnxn$ , where  $f$  is frequency axis and  $n$  is number of ports. Note that indexing starts at 0, so `s11` can be accessed by taking the slice `s[:,0,0]`.

**Returns** `s` : complex `numpy.ndarray` of shape  $fnxn$   
the scattering parameter matrix.

#### See Also:

`s`, `y`, `z`, `t`, `a`

## References

### skrf.network.Network.s11

Network.**s11**

one-port sub-network.

### skrf.network.Network.s12

Network.**s12**

one-port sub-network.

### skrf.network.Network.s21

Network.**s21**

one-port sub-network.

### skrf.network.Network.s22

Network.**s22**

one-port sub-network.

### skrf.network.Network.s\_arcl

Network.**s\_arcl**

The `arcl` component of the `s`-matrix

#### See Also:

`s`

**skrf.network.Network.s\_arcl\_unwrap**`Network.s_arcl_unwrap`

The arcl\_unwrap component of the s-matrix

**See Also:**

s

**skrf.network.Network.s\_db**`Network.s_db`

The db component of the s-matrix

**See Also:**

s

**skrf.network.Network.s\_deg**`Network.s_deg`

The deg component of the s-matrix

**See Also:**

s

**skrf.network.Network.s\_deg\_unwrap**`Network.s_deg_unwrap`

The deg\_unwrap component of the s-matrix

**See Also:**

s

**skrf.network.Network.s\_im**`Network.s_im`

The im component of the s-matrix

**See Also:**

s

**skrf.network.Network.s\_mag**`Network.s_mag`

The mag component of the s-matrix

**See Also:**

s

### skrf.network.Network.s\_rad

#### Network.s\_rad

The rad component of the s-matrix

#### See Also:

s

### skrf.network.Network.s\_rad\_unwrap

#### Network.s\_rad\_unwrap

The rad\_unwrap component of the s-matrix

#### See Also:

s

### skrf.network.Network.s\_re

#### Network.s\_re

The re component of the s-matrix

#### See Also:

s

### skrf.network.Network.t

#### Network.t

Scattering transfer parameters

The t-matrix<sup>3</sup> is a 3-dimensional `numpy.ndarray` which has shape  $f \times 2 \times 2$ , where  $f$  is frequency axis. Note that indexing starts at 0, so t11 can be accessed by taking the slice  $t[:,0,0]$ .

The t-matrix, also known as the wave cascading matrix, is only defined for a 2-port Network.

**Returns** **t** : complex `numpy.ndarray` of shape  $f \times 2 \times 2$

t-parameters, aka scattering transfer parameters

#### See Also:

s, y, z, t, a

#### References

### skrf.network.Network.y

#### Network.y

Admittance parameter matrix.

The y-matrix<sup>4</sup> is a 3-dimensional `numpy.ndarray` which has shape  $f \times n \times n$ , where  $f$  is frequency axis and  $n$  is number of ports. Note that indexing starts at 0, so y11 can be accessed by taking the slice  $y[:,0,0]$ .

---

<sup>3</sup> [http://en.wikipedia.org/wiki/Scattering\\_parameters#Scattering\\_transfer\\_parameters](http://en.wikipedia.org/wiki/Scattering_parameters#Scattering_transfer_parameters)

<sup>4</sup> [http://en.wikipedia.org/wiki/Admittance\\_parameters](http://en.wikipedia.org/wiki/Admittance_parameters)

**Returns** `y` : complex `numpy.ndarray` of shape `fxnxn`  
the admittance parameter matrix.

**See Also:**

`s`, `y`, `z`, `t`, `a`

**References****skrf.network.Network.y\_arcl**`Network.y_arcl`

The arcl component of the y-matrix

**See Also:**

`Y`

**skrf.network.Network.y\_arcl\_unwrap**`Network.y_arcl_unwrap`

The arcl\_unwrap component of the y-matrix

**See Also:**

`Y`

**skrf.network.Network.y\_db**`Network.y_db`

The db component of the y-matrix

**See Also:**

`Y`

**skrf.network.Network.y\_deg**`Network.y_deg`

The deg component of the y-matrix

**See Also:**

`Y`

**skrf.network.Network.y\_deg\_unwrap**`Network.y_deg_unwrap`

The deg\_unwrap component of the y-matrix

**See Also:**

`Y`

### `skrf.network.Network.y_im`

`Network.y_im`

The im component of the y-matrix

**See Also:**

`Y`

### `skrf.network.Network.y_mag`

`Network.y_mag`

The mag component of the y-matrix

**See Also:**

`Y`

### `skrf.network.Network.y_rad`

`Network.y_rad`

The rad component of the y-matrix

**See Also:**

`Y`

### `skrf.network.Network.y_rad_unwrap`

`Network.y_rad_unwrap`

The rad\_unwrap component of the y-matrix

**See Also:**

`Y`

### `skrf.network.Network.y_re`

`Network.y_re`

The re component of the y-matrix

**See Also:**

`Y`

### `skrf.network.Network.z`

`Network.z`

Impedance parameter matrix.

The z-matrix<sup>5</sup> is a 3-dimensional `numpy.ndarray` which has shape  $f \times n \times n$ , where  $f$  is frequency axis and  $n$  is number of ports. Note that indexing starts at 0, so `z[1,1]` can be accessed by taking the slice `z[:,0,0]`.

**Returns** `z`: complex `numpy.ndarray` of shape  $f \times n \times n$

---

<sup>5</sup> [http://en.wikipedia.org/wiki/impedance\\_parameters](http://en.wikipedia.org/wiki/impedance_parameters)



the Impedance parameter matrix.

**See Also:**

`s`, `y`, `z`, `t`, `a`

**References**

**skrf.network.Network.z0**

`Network.z0`

Characteristic impedance[s] of the network ports.

This property stores the characteristic impedance of each port of the network. Because it is possible that each port has a different characteristic impedance each varying with frequency, `z0` is stored internally as a *fxn* array.

However because `z0` is frequently simple (like 50ohm), it can be set with just number as well.

**Returns** `z0`: `numpy.ndarray` of shape *fxn*  
characteristic impedance for network

**skrf.network.Network.z\_arcl**

`Network.z_arcl`

The arcl component of the z-matrix

**See Also:**

`z`

**skrf.network.Network.z\_arcl\_unwrap**

`Network.z_arcl_unwrap`

The arcl\_unwrap component of the z-matrix

**See Also:**

`z`

**skrf.network.Network.z\_db**

`Network.z_db`

The db component of the z-matrix

**See Also:**

`z`

**skrf.network.Network.z\_deg**

`Network.z_deg`

The deg component of the z-matrix

**See Also:**

z

### **skrf.network.Network.z\_deg\_unwrap**

`Network.z_deg_unwrap`

The `deg_unwrap` component of the z-matrix

**See Also:**

z

### **skrf.network.Network.z\_im**

`Network.z_im`

The `im` component of the z-matrix

**See Also:**

z

### **skrf.network.Network.z\_mag**

`Network.z_mag`

The `mag` component of the z-matrix

**See Also:**

z

### **skrf.network.Network.z\_rad**

`Network.z_rad`

The `rad` component of the z-matrix

**See Also:**

z

### **skrf.network.Network.z\_rad\_unwrap**

`Network.z_rad_unwrap`

The `rad_unwrap` component of the z-matrix

**See Also:**

z

### **skrf.network.Network.z\_re**

`Network.z_re`

The `re` component of the z-matrix

**See Also:**

z

## Methods

<code>__init__</code>	Network constructor.
<code>add_noise_polar</code>	adds a complex zero-mean gaussian white-noise.
<code>add_noise_polar_flatband</code>	adds a flatband complex zero-mean gaussian white-noise signal of
<code>copy</code>	Returns a copy of this Network
<code>copy_from</code>	Copies the contents of another Network into self
<code>flip</code>	swaps the ports of a two port Network
<code>interpolate</code>	Return an interpolated network, from a new :class:`~skrf.frequency.Frequency`.
<code>interpolate_from_f</code>	Interpolates s-parameters from a frequency vector.
<code>interpolate_self</code>	Interpolates s-parameters given a new
<code>interpolate_self_npoints</code>	Interpolate network based on a new number of frequency points
<code>multiply_noise</code>	multiplies a complex bivariate gaussian white-noise signal
<code>nudge</code>	Perturb s-parameters by small amount.
<code>plot_a_arcl</code>	plot the Network attribute <code>a_arcl</code> vs frequency.
<code>plot_a_arcl_unwrap</code>	plot the Network attribute <code>a_arcl_unwrap</code> vs frequency.
<code>plot_a_complex</code>	plot the Network attribute <code>a</code> vs frequency.
<code>plot_a_db</code>	plot the Network attribute <code>a_db</code> vs frequency.
<code>plot_a_deg</code>	plot the Network attribute <code>a_deg</code> vs frequency.
<code>plot_a_deg_unwrap</code>	plot the Network attribute <code>a_deg_unwrap</code> vs frequency.
<code>plot_a_im</code>	plot the Network attribute <code>a_im</code> vs frequency.
<code>plot_a_mag</code>	plot the Network attribute <code>a_mag</code> vs frequency.
<code>plot_a_polar</code>	plot the Network attribute <code>a</code> vs frequency.
<code>plot_a_rad</code>	plot the Network attribute <code>a_rad</code> vs frequency.
<code>plot_a_rad_unwrap</code>	plot the Network attribute <code>a_rad_unwrap</code> vs frequency.
<code>plot_a_re</code>	plot the Network attribute <code>a_re</code> vs frequency.
<code>plot_it_all</code>	
<code>plot_passivity</code>	plots the passivity of a network, possibly for a specific port.
<code>plot_s_arcl</code>	plot the Network attribute <code>s_arcl</code> vs frequency.
<code>plot_s_arcl_unwrap</code>	plot the Network attribute <code>s_arcl_unwrap</code> vs frequency.
<code>plot_s_complex</code>	plot the Network attribute <code>s</code> vs frequency.
<code>plot_s_db</code>	plot the Network attribute <code>s_db</code> vs frequency.
<code>plot_s_deg</code>	plot the Network attribute <code>s_deg</code> vs frequency.
<code>plot_s_deg_unwrap</code>	plot the Network attribute <code>s_deg_unwrap</code> vs frequency.
<code>plot_s_im</code>	plot the Network attribute <code>s_im</code> vs frequency.
<code>plot_s_mag</code>	plot the Network attribute <code>s_mag</code> vs frequency.
<code>plot_s_polar</code>	plot the Network attribute <code>s</code> vs frequency.
<code>plot_s_rad</code>	plot the Network attribute <code>s_rad</code> vs frequency.
<code>plot_s_rad_unwrap</code>	plot the Network attribute <code>s_rad_unwrap</code> vs frequency.
<code>plot_s_re</code>	plot the Network attribute <code>s_re</code> vs frequency.
<code>plot_s_smith</code>	plots the scattering parameter on a smith chart
<code>plot_y_arcl</code>	plot the Network attribute <code>y_arcl</code> vs frequency.
<code>plot_y_arcl_unwrap</code>	plot the Network attribute <code>y_arcl_unwrap</code> vs frequency.
<code>plot_y_complex</code>	plot the Network attribute <code>y</code> vs frequency.
<code>plot_y_db</code>	plot the Network attribute <code>y_db</code> vs frequency.
<code>plot_y_deg</code>	plot the Network attribute <code>y_deg</code> vs frequency.
<code>plot_y_deg_unwrap</code>	plot the Network attribute <code>y_deg_unwrap</code> vs frequency.
<code>plot_y_im</code>	plot the Network attribute <code>y_im</code> vs frequency.
<code>plot_y_mag</code>	plot the Network attribute <code>y_mag</code> vs frequency.
<code>plot_y_polar</code>	plot the Network attribute <code>y</code> vs frequency.

Continued on next page

**Table 3.6 – continued from previous page**

<code>plot_y_rad</code>	plot the Network attribute <code>y_rad</code> vs frequency.
<code>plot_y_rad_unwrap</code>	plot the Network attribute <code>y_rad_unwrap</code> vs frequency.
<code>plot_y_re</code>	plot the Network attribute <code>y_re</code> vs frequency.
<code>plot_z_arcl</code>	plot the Network attribute <code>z_arcl</code> vs frequency.
<code>plot_z_arcl_unwrap</code>	plot the Network attribute <code>z_arcl_unwrap</code> vs frequency.
<code>plot_z_complex</code>	plot the Network attribute <code>z</code> vs frequency.
<code>plot_z_db</code>	plot the Network attribute <code>z_db</code> vs frequency.
<code>plot_z_deg</code>	plot the Network attribute <code>z_deg</code> vs frequency.
<code>plot_z_deg_unwrap</code>	plot the Network attribute <code>z_deg_unwrap</code> vs frequency.
<code>plot_z_im</code>	plot the Network attribute <code>z_im</code> vs frequency.
<code>plot_z_mag</code>	plot the Network attribute <code>z_mag</code> vs frequency.
<code>plot_z_polar</code>	plot the Network attribute <code>z</code> vs frequency.
<code>plot_z_rad</code>	plot the Network attribute <code>z_rad</code> vs frequency.
<code>plot_z_rad_unwrap</code>	plot the Network attribute <code>z_rad_unwrap</code> vs frequency.
<code>plot_z_re</code>	plot the Network attribute <code>z_re</code> vs frequency.
<code>read</code>	Read a Network from a 'ntwk' file
<code>read_touchstone</code>	loads values from a touchstone file.
<code>renumber</code>	renumbers some ports of a two port Network
<code>resample</code>	Interpolate network based on a new number of frequency points
<code>write</code>	Write the Network to disk using the <code>pickle</code> module.
<code>write_touchstone</code>	write a contents of the Network to a touchstone file.

### `skrf.network.Network.__init__`

`Network.__init__` (*file=None, name=None, comments=None, \*\*kwargs*)

Network constructor.

Creates an n-port microwave network from a *file* or directly from data. If no file or data is given, then an empty Network is created.

**Parameters** `file` : str or file-object

**file to load information from. supported formats are:**

- touchstone file (.s?p)
- pickled Network (.ntwk, .p) see `write()`

**name** : str

Name of this Network. if None will try to use file, if its a str

**comments** : str

Comments associated with the Network

**\*\*kwargs** : :

key word arguments can be used to assign properties of the Network, such as *s*, *f* and *z0*.

### See Also:

`read` read a network from a file

`write` write a network to a file, using pickle

`write_touchstone` write a network to a touchstone file

## Examples

From a touchstone

```
>>> n = rf.Network('ntwk1.s2p')
```

From a pickle file

```
>>> n = rf.Network('ntwk1.ntwk')
```

Create a blank network, then fill in values

```
>>> n = rf.Network()
>>> n.f, n.s, n.z0 = [1,2,3], [1,2,3], [1,2,3]
```

Directly from values

```
>>> n = rf.Network(f=[1,2,3], s=[1,2,3], z0=[1,2,3])
```

### skrf.network.Network.add\_noise\_polar

`Network.add_noise_polar` (*mag\_dev*, *phase\_dev*, *\*\*kwargs*)  
adds a complex zero-mean gaussian white-noise.

adds a complex zero-mean gaussian white-noise of a given standard deviation for magnitude and phase

**Parameters** `mag_dev` : number

standard deviation of magnitude

`phase_dev` : number

standard deviation of phase [in degrees]

### skrf.network.Network.add\_noise\_polar\_flatband

`Network.add_noise_polar_flatband` (*mag\_dev*, *phase\_dev*, *\*\*kwargs*)

adds a flatband complex zero-mean gaussian white-noise signal of given standard deviations for magnitude and phase

**Parameters** `mag_dev` : number

standard deviation of magnitude

`phase_dev` : number

standard deviation of phase [in degrees]

### skrf.network.Network.copy

`Network.copy` ()

Returns a copy of this Network

Needed to allow pass-by-value for a Network instead of pass-by-reference

### skrf.network.Network.copy\_from

Network.**copy\_from**(*other*)

Copies the contents of another Network into self

Uses copy, so that the data is passed-by-value, not reference

**Parameters** **other** : Network

the network to copy the contents of

#### Examples

```
>>> a = rf.N()
>>> b = rf.N('my_file.s2p')
>>> a.copy_from (b)
```

### skrf.network.Network.flip

Network.**flip**()

swaps the ports of a two port Network

### skrf.network.Network.interpolate

Network.**interpolate**(*new\_frequency*, **\*\*kwargs**)

Return an interpolated network, from a new :class:`~skrf.frequency.Frequency`.

Interpolate the networks s-parameters linearly in real and imaginary components. Other interpolation types can be used by passing appropriate **\*\*kwargs**. This function *returns* an interpolated Network. Alternatively `interpolate_self()` will interpolate self.

**Parameters** **new\_frequency** : Frequency

frequency information to interpolate

**\*\*kwargs** : keyword arguments

passed to `scipy.interpolate.interpld()` initializer.

**Returns** **result** : Network

an interpolated Network

#### See Also:

`resample`, `interpolate_self`, `interpolate_from_f`

#### Notes

See `scipy.interpolate.interpolate.interpld()` for useful kwargs. For example

**kind** [str or int] Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic') or as an integer specifying the order of the spline interpolator to use.

## Examples

```
In [2]: n = rf.data.ring_slot
```

```
In [3]: n
```

```
Out[3]: 2-Port Network: 'ring slot', 75-110 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j]
```

```
In [4]: new_freq = rf.Frequency(75,110,501,'ghz')
```

```
In [5]: n.interpolate(new_freq, kind = 'cubic')
```

```
Out[5]: 2-Port Network: 'ring slot', 75-110 GHz, 501 pts, z0=[ 50.+0.j 50.+0.j]
```

### skrf.network.Network.interpolate\_from\_f

`Network.interpolate_from_f(f, interp_kwargs={}, **kwargs)`

Interpolates s-parameters from a frequency vector.

Given a frequency vector, and optionally a *unit* (see `**kwargs`), interpolate the networks s-parameters linearly in real and imaginary components.

See `interpolate()` for more information.

**Parameters** `new_frequency`: `Frequency`

frequency information to interpolate at

**interp\_kwargs** :

dictionary of kwargs to be passed through to `scipy.interpolate.interp1d()`

**\*\*kwargs** :

passed to `scipy.interpolate.interp1d()` initializer.

**See Also:**

`resample`, `interpolate`, `interpolate_self`

### Notes

This creates a new `Frequency` object using the method `from_f()`, and then calls `interpolate_self()`.

### skrf.network.Network.interpolate\_self

`Network.interpolate_self(new_frequency, **kwargs)`

Interpolates s-parameters given a new :class:`~skrf.frequency.Frequency` object.

See `interpolate()` for more information.

**Parameters** `new_frequency`: `Frequency`

frequency information to interpolate at

**\*\*kwargs**: keyword arguments

passed to `scipy.interpolate.interp1d()` initializer.

**See Also:**

`resample`, `interpolate`, `interpolate_from_f`

**skrf.network.Network.interpolate\_self\_npoints**

`Network.interpolate_self_npoints` (*npoints*, *\*\*kwargs*)  
Interpolate network based on a new number of frequency points

**Parameters** `npoints` : int

number of frequency points

**\*\*kwargs** : keyword arguments

passed to `scipy.interpolate.interpld()` initializer.

**See Also:**

`interpolate_self` same functionality but takes a Frequency object

`interpolate` same functionality but takes a Frequency object and returns a new Network, instead of updating itself.

**Notes**

The function `resample()` is an alias for `interpolate_self_npoints()`.

**Examples**

```
In [2]: n = rf.data.ring_slot
```

```
In [3]: n
```

```
Out[3]: 2-Port Network: 'ring slot', 75-110 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j]
```

```
In [4]: n.resample(501) # resample is an alias
```

```
In [5]: n
```

```
Out[5]: 2-Port Network: 'ring slot', 75-110 GHz, 501 pts, z0=[ 50.+0.j 50.+0.j]
```

**skrf.network.Network.multiply\_noise**

`Network.multiply_noise` (*mag\_dev*, *phase\_dev*, *\*\*kwargs*)

multiplies a complex bivariate gaussian white-noise signal of given standard deviations for magnitude and phase. magnitude mean is 1, phase mean is 0

**takes:** `mag_dev`: standard deviation of magnitude `phase_dev`: standard deviation of phase [in degrees] `n_ports`: number of ports. default to 1

**returns:** nothing



**skrf.network.Network.nudge**

`Network.nudge` (*amount=1e-12*)

Perturb s-parameters by small amount.

This is useful to work-around numerical bugs.

**Parameters** **amount** : number,  
amount to add to s parameters

**Notes**

This function is `self.s = self.s + 1e-12`

**skrf.network.Network.plot\_a\_arcl**

`Network.plot_a_arcl` (*m=None, n=None, ax=None, show\_legend=True, attribute='a\_arcl', y\_label='Arc Length', \*args, \*\*kwargs*)

plot the Network attribute `a_arcl` vs frequency.

**Parameters** **m** : int, optional  
first index of s-parameter matrix, if None will use all

**n** : int, optional  
secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional  
An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_a_arcl(m=1,n=0,color='r')
```

**skrf.network.Network.plot\_a\_arcl\_unwrap**

`Network.plot_a_arcl_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='a\_arcl\_unwrap'*, *y\_label='Arc Length'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `a_arcl_unwrap` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_a_arcl_unwrap(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_a\_complex**

`Network.plot_a_complex` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *prop\_name='a'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `a` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_a(m=1,n=0,color='r')
```

### `skrf.network.Network.plot_a_db`

`Network.plot_a_db(m=None, n=None, ax=None, show_legend=True, attribute='a_db', y_label='Magnitude (dB)', *args, **kwargs)`  
plot the Network attribute `a_db` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_a_db(m=1,n=0,color='r')
```

## `skrf.network.Network.plot_a_deg`

`Network.plot_a_deg(m=None, n=None, ax=None, show_legend=True, attribute='a_deg', y_label='Phase (deg)', *args, **kwargs)`  
plot the Network attribute `a_deg` vs frequency.

**Parameters** `m` : int, optional

first index of s-parameter matrix, if None will use all

`n` : int, optional

secon index of the s-parameter matrix, if None will use all

`ax` : `matplotlib.Axes` object, optional

An existing Axes object to plot on

`show_legend` : Boolean

draw legend or not

`attribute` : string

Network attribute to plot

`y_label` : string, optional

the y-axis label

`*args, **kwargs` : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_a_deg(m=1,n=0,color='r')
```

**skrf.network.Network.plot\_a\_deg\_unwrap**

`Network.plot_a_deg_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='a\_deg\_unwrap'*, *y\_label='Phase (deg)'*, \*args, \*\*kwargs)  
 plot the Network attribute `a_deg_unwrap` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_a_deg_unwrap(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_a\_im**

`Network.plot_a_im` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='a\_im'*, *y\_label='Imag Part'*, \*args, \*\*kwargs)  
 plot the Network attribute `a_im` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_a_im(m=1,n=0,color='r')
```

### `skrf.network.Network.plot_a_mag`

`Network.plot_a_mag(m=None, n=None, ax=None, show_legend=True, attribute='a_mag', y_label='Magnitude', *args, **kwargs)`  
plot the Network attribute `a_mag` vs frequency.

**Parameters** **m** : int, optional  
first index of s-parameter matrix, if None will use all

**n** : int, optional  
second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional  
An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_a_mag(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_a_polar`

`Network.plot_a_polar` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *prop\_name='a'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `a` vs frequency.

**Parameters** `m` : int, optional

first index of s-parameter matrix, if None will use all

`n` : int, optional

second index of the s-parameter matrix, if None will use all

`ax` : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_a(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_a\_rad**

`Network.plot_a_rad(m=None, n=None, ax=None, show_legend=True, attribute='a_rad', y_label='Phase (rad)', *args, **kwargs)`  
plot the Network attribute `a_rad` vs frequency.

**Parameters** `m` : int, optional

first index of s-parameter matrix, if None will use all

`n` : int, optional

second index of the s-parameter matrix, if None will use all

`ax` : `matplotlib.Axes` object, optional

An existing Axes object to plot on

`show_legend` : Boolean

draw legend or not

`attribute` : string

Network attribute to plot

`y_label` : string, optional

the y-axis label

`*args, **kwargs` : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_a_rad(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_a\_rad\_unwrap**

`Network.plot_a_rad_unwrap(m=None, n=None, ax=None, show_legend=True, attribute='a_rad_unwrap', y_label='Phase (rad)', *args, **kwargs)`  
plot the Network attribute `a_rad_unwrap` vs frequency.

**Parameters** `m` : int, optional

first index of s-parameter matrix, if None will use all

`n` : int, optional

second index of the s-parameter matrix, if None will use all

`ax` : `matplotlib.Axes` object, optional

An existing Axes object to plot on



**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_a_rad_unwrap(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_a_re`

`Network.plot_a_re` (*m=None, n=None, ax=None, show\_legend=True, attribute='a\_re', y\_label='Real Part', \*args, \*\*kwargs*)  
plot the Network attribute `a_re` vs frequency.

**Parameters** **m** : int, optional  
first index of s-parameter matrix, if None will use all

**n** : int, optional  
second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional  
An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_a_re(m=1,n=0,color='r')
```

### `skrf.network.Network.plot_it_all`

`Network.plot_it_all(*args, **kwargs)`

### `skrf.network.Network.plot_passivity`

`Network.plot_passivity(port=None, ax=None, show_legend=True, *args, **kwargs)`  
plots the passivity of a network, possibly for a specific port.

**Parameters** `port: int` :

calculate passivity of a given port

**ax** : matplotlib.Axes object, optional

axes to plot on. in case you want to update an existing plot.

**show\_legend** : boolean, optional

to turn legend show legend of not, optional

**\*args** : arguments, optional

passed to the matplotlib.plot command

**\*\*kwargs** : keyword arguments, optional

passed to the matplotlib.plot command

**See Also:**

`plot_vs_frequency_generic`, `passivity`

## Examples

```
>>> myntwk.plot_s_rad()
>>> myntwk.plot_s_rad(m=0,n=1,color='b', marker='x')
```

### `skrf.network.Network.plot_s_arcl`

`Network.plot_s_arcl(m=None, n=None, ax=None, show_legend=True, attribute='s_arcl', y_label='Arc Length', *args, **kwargs)`  
plot the Network attribute `s_arcl` vs frequency.

**Parameters** `m` : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional  
 secon index of the s-parameter matrix, if None will use all

**ax** : matplotlib.Axes object, optional  
 An existing Axes object to plot on

**show\_legend** : Boolean  
 draw legend or not

**attribute** : string  
 Network attribute to plot

**y\_label** : string, optional  
 the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
 passed to matplotlib.plot()

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_s_arcl(m=1,n=0,color='r')
```

### skrf.network.Network.plot\_s\_arcl\_unwrap

`Network.plot_s_arcl_unwrap(m=None, n=None, ax=None, show_legend=True, attribute='s_arcl_unwrap', y_label='Arc Length', *args, **kwargs)`  
 plot the Network attribute `s_arcl_unwrap` vs frequency.

**Parameters** **m** : int, optional  
 first index of s-parameter matrix, if None will use all

**n** : int, optional  
 secon index of the s-parameter matrix, if None will use all

**ax** : matplotlib.Axes object, optional  
 An existing Axes object to plot on

**show\_legend** : Boolean  
 draw legend or not

**attribute** : string  
 Network attribute to plot

**y\_label** : string, optional  
 the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_s_arcl_unwrap(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_s_complex`

`Network.plot_s_complex` (*m=None, n=None, ax=None, show\_legend=True, prop\_name='s', \*args, \*\*kwargs*)  
plot the Network attribute `s` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_s(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_s\_db**

`Network.plot_s_db` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='s\_db'*,  
*y\_label='Magnitude (dB)'*, \*args, \*\*kwargs)  
 plot the Network attribute `s_db` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_s_db(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_s\_deg**

`Network.plot_s_deg` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='s\_deg'*,  
*y\_label='Phase (deg)'*, \*args, \*\*kwargs)  
 plot the Network attribute `s_deg` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_s_deg(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_s_deg_unwrap`

`Network.plot_s_deg_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='s\_deg\_unwrap'*, *y\_label='Phase (deg)'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `s_deg_unwrap` vs frequency.

**Parameters** **m** : int, optional  
first index of s-parameter matrix, if None will use all

**n** : int, optional  
secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional  
An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_s_deg_unwrap(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_s_im`

`Network.plot_s_im` (*m=None, n=None, ax=None, show\_legend=True, attribute='s\_im', y\_label='Imag Part', \*args, \*\*kwargs*)  
plot the Network attribute `s_im` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_s_im(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_s\_mag**

`Network.plot_s_mag` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='s\_mag'*,  
*y\_label='Magnitude'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `s_mag` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_s_mag(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_s\_polar**

`Network.plot_s_polar` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *prop\_name='s'*, *\*args*,  
*\*\*kwargs*)  
plot the Network attribute `s` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on



**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_s(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_s_rad`

`Network.plot_s_rad(m=None, n=None, ax=None, show_legend=True, attribute='s_rad', y_label='Phase (rad)', *args, **kwargs)`  
plot the Network attribute `s_rad` vs frequency.

**Parameters** **m** : int, optional  
first index of s-parameter matrix, if None will use all

**n** : int, optional  
second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional  
An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_s_rad(m=1, n=0, color='r')
```

## `skrf.network.Network.plot_s_rad_unwrap`

`Network.plot_s_rad_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='s\_rad\_unwrap'*, *y\_label='Phase (rad)'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `s_rad_unwrap` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_s_rad_unwrap(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_s\_re**

`Network.plot_s_re` (*m=None, n=None, ax=None, show\_legend=True, attribute='s\_re', y\_label='Real Part', \*args, \*\*kwargs*)

plot the Network attribute `s_re` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_s_re(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_s\_smith**

`Network.plot_s_smith` (*m=None, n=None, r=1, ax=None, show\_legend=True, chart\_type='z', draw\_labels=False, label\_axes=False, \*args, \*\*kwargs*)

plots the scattering parameter on a smith chart

plots indices *m, n*, where *m* and *n* can be integers or lists of integers.

**Parameters** **m** : int, optional

first index

**n** : int, optional

second index

**ax** : `matplotlib.Axes` object, optional

axes to plot on. in case you want to update an existing plot.

**show\_legend** : boolean, optional

to turn legend show legend of not, optional

**chart\_type** : ['z','y']

draw impedance or admittance contours

**draw\_labels** : Boolean

annotate chart with impedance values

**label\_axes** : Boolean

Label axis with titles *Real* and *Imaginary*

**border** : Boolean

draw rectangular border around image with ticks

**\*args** : arguments, optional

passed to the matplotlib.plot command

**\*\*kwargs** : keyword arguments, optional

passed to the matplotlib.plot command

**See Also:**

`plot_vs_frequency_generic`, `smith`

**Examples**

```
>>> myntwk.plot_s_smith()
>>> myntwk.plot_s_smith(m=0,n=1,color='b', marker='x')
```

**skrf.network.Network.plot\_y\_arcl**

`Network.plot_y_arcl` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='y\_arcl'*, *y\_label='Arc Length'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `y_arcl` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : matplotlib.Axes object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_y_arcl(m=1,n=0,color='r')
```

### `skrf.network.Network.plot_y_arcl_unwrap`

`Network.plot_y_arcl_unwrap` (*m=None, n=None, ax=None, show\_legend=True, attribute='y\_arcl\_unwrap', y\_label='Arc Length', \*args, \*\*kwargs*)  
plot the Network attribute `y_arcl_unwrap` vs frequency.

**Parameters** **m** : int, optional  
first index of s-parameter matrix, if None will use all

**n** : int, optional  
secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional  
An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_y_arcl_unwrap(m=1, n=0, color='r')
```

## skrf.network.Network.plot\_y\_complex

`Network.plot_y_complex` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *prop\_name='y'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `y` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_y(m=1, n=0, color='r')
```

## skrf.network.Network.plot\_y\_db

`Network.plot_y_db` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='y\_db'*, *y\_label='Magnitude (dB)'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `y_db` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

secon index of the s-parameter matrix, if None will use all  
**ax** : matplotlib.Axes object, optional  
 An existing Axes object to plot on  
**show\_legend** : Boolean  
 draw legend or not  
**attribute** : string  
 Network attribute to plot  
**y\_label** : string, optional  
 the y-axis label  
**\*args,\*\*kwargs** : arguments, keyword arguments  
 passed to matplotlib.plot()

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_y_db(m=1, n=0, color='r')
```

### skrf.network.Network.plot\_y\_deg

`Network.plot_y_deg(m=None, n=None, ax=None, show_legend=True, attribute='y_deg', y_label='Phase (deg)', *args, **kwargs)`  
 plot the Network attribute `y_deg` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

secon index of the s-parameter matrix, if None will use all

**ax** : matplotlib.Axes object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_y_deg(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_y_deg_unwrap`

`Network.plot_y_deg_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='y\_deg\_unwrap'*, *y\_label='Phase (deg)'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `y_deg_unwrap` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_y_deg_unwrap(m=1, n=0, color='r')
```



**skrf.network.Network.plot\_y\_im**

`Network.plot_y_im` (*m=None, n=None, ax=None, show\_legend=True, attribute='y\_im', y\_label='Imag Part', \*args, \*\*kwargs*)  
 plot the Network attribute `y_im` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_y_im(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_y\_mag**

`Network.plot_y_mag` (*m=None, n=None, ax=None, show\_legend=True, attribute='y\_mag', y\_label='Magnitude', \*args, \*\*kwargs*)  
 plot the Network attribute `y_mag` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_y_mag(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_y_polar`

`Network.plot_y_polar` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *prop\_name='y'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `y` vs frequency.

**Parameters** **m** : int, optional  
first index of s-parameter matrix, if None will use all

**n** : int, optional  
second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional  
An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_y(m=1,n=0,color='r')
```

### `skrf.network.Network.plot_y_rad`

`Network.plot_y_rad(m=None, n=None, ax=None, show_legend=True, attribute='y_rad', y_label='Phase (rad)', *args, **kwargs)`  
plot the Network attribute `y_rad` vs frequency.

**Parameters** `m` : int, optional

first index of s-parameter matrix, if None will use all

`n` : int, optional

second index of the s-parameter matrix, if None will use all

`ax` : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_y_rad(m=1,n=0,color='r')
```

**skrf.network.Network.plot\_y\_rad\_unwrap**

`Network.plot_y_rad_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='y\_rad\_unwrap'*, *y\_label='Phase (rad)'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `y_rad_unwrap` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_y_rad_unwrap(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_y\_re**

`Network.plot_y_re` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='y\_re'*, *y\_label='Real Part'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `y_re` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_y_re(m=1,n=0,color='r')
```

### `skrf.network.Network.plot_z_arcl`

`Network.plot_z_arcl(m=None, n=None, ax=None, show_legend=True, attribute='z_arcl', y_label='Arc Length', *args, **kwargs)`  
plot the Network attribute `z_arcl` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_z_arcl(m=1,n=0,color='r')
```

## `skrf.network.Network.plot_z_arcl_unwrap`

`Network.plot_z_arcl_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='z\_arcl\_unwrap'*, *y\_label='Arc Length'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute `z_arcl_unwrap` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_z_arcl_unwrap(m=1,n=0,color='r')
```

**skrf.network.Network.plot\_z\_complex**

`Network.plot_z_complex` (*m=None, n=None, ax=None, show\_legend=True, prop\_name='z', \*args, \*\*kwargs*)

plot the Network attribute *z* vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_z(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_z\_db**

`Network.plot_z_db` (*m=None, n=None, ax=None, show\_legend=True, attribute='z\_db', y\_label='Magnitude (dB)', \*args, \*\*kwargs*)

plot the Network attribute *z\_db* vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_z_db(m=1,n=0,color='r')
```

### `skrf.network.Network.plot_z_deg`

`Network.plot_z_deg(m=None, n=None, ax=None, show_legend=True, attribute='z_deg', y_label='Phase (deg)', *args, **kwargs)`  
plot the Network attribute `z_deg` vs frequency.

**Parameters** **m** : int, optional  
first index of s-parameter matrix, if None will use all

**n** : int, optional  
secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional  
An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`



## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_z_deg(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_z_deg_unwrap`

`Network.plot_z_deg_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='z\_deg\_unwrap'*, *y\_label='Phase (deg)'*, \*args, \*\*kwargs) plot the Network attribute `z_deg_unwrap` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_z_deg_unwrap(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_z\_im**

`Network.plot_z_im` (*m=None, n=None, ax=None, show\_legend=True, attribute='z\_im', y\_label='Imag Part', \*args, \*\*kwargs*)  
plot the Network attribute `z_im` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_z_im(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_z\_mag**

`Network.plot_z_mag` (*m=None, n=None, ax=None, show\_legend=True, attribute='z\_mag', y\_label='Magnitude', \*args, \*\*kwargs*)  
plot the Network attribute `z_mag` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

secon index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_z_mag(m=1, n=0, color='r')
```

### `skrf.network.Network.plot_z_polar`

`Network.plot_z_polar` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *prop\_name='z'*, *\*args*, *\*\*kwargs*)  
plot the Network attribute *z* vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_z(m=1,n=0,color='r')
```

## `skrf.network.Network.plot_z_rad`

`Network.plot_z_rad(m=None, n=None, ax=None, show_legend=True, attribute='z_rad', y_label='Phase (rad)', *args, **kwargs)`  
plot the Network attribute `z_rad` vs frequency.

**Parameters** `m` : int, optional

first index of s-parameter matrix, if None will use all

`n` : int, optional

second index of the s-parameter matrix, if None will use all

`ax` : `matplotlib.Axes` object, optional

An existing Axes object to plot on

`show_legend` : Boolean

draw legend or not

`attribute` : string

Network attribute to plot

`y_label` : string, optional

the y-axis label

`*args, **kwargs` : arguments, keyword arguments

passed to `matplotlib.plot()`

## Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

## Examples

```
>>> myntwk.plot_z_rad(m=1,n=0,color='r')
```

**skrf.network.Network.plot\_z\_rad\_unwrap**

`Network.plot_z_rad_unwrap` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='z\_rad\_unwrap'*, *y\_label='Phase (rad)'*, *\*args*, *\*\*kwargs*)  
 plot the Network attribute `z_rad_unwrap` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean

draw legend or not

**attribute** : string

Network attribute to plot

**y\_label** : string, optional

the y-axis label

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to `matplotlib.plot()`

**Notes**

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

**Examples**

```
>>> myntwk.plot_z_rad_unwrap(m=1, n=0, color='r')
```

**skrf.network.Network.plot\_z\_re**

`Network.plot_z_re` (*m=None*, *n=None*, *ax=None*, *show\_legend=True*, *attribute='z\_re'*, *y\_label='Real Part'*, *\*args*, *\*\*kwargs*)  
 plot the Network attribute `z_re` vs frequency.

**Parameters** **m** : int, optional

first index of s-parameter matrix, if None will use all

**n** : int, optional

second index of the s-parameter matrix, if None will use all

**ax** : `matplotlib.Axes` object, optional

An existing Axes object to plot on

**show\_legend** : Boolean  
draw legend or not

**attribute** : string  
Network attribute to plot

**y\_label** : string, optional  
the y-axis label

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `matplotlib.plot()`

### Notes

This function is dynamically generated upon Network initialization. This is accomplished by calling `plot_vs_frequency_generic()`

### Examples

```
>>> myntwk.plot_z_re(m=1,n=0,color='r')
```

### `skrf.network.Network.read`

`Network.read(*args, **kwargs)`

Read a Network from a 'ntwk' file

A ntwk file is written with `write()`. It is just a pickled file.

**Parameters** **\*args,\*\*kwargs** : args and kwargs  
passed to `skrf.io.general.write()`

### See Also:

`write`, `skrf.io.general.write`, `skrf.io.general.read`

### Notes

This function calls `skrf.io.general.read()`.

### Examples

```
>>> rf.read('myfile.ntwk')
>>> rf.read('myfile.p')
```

**skrf.network.Network.read\_touchstone**

`Network.read_touchstone` (*filename*)

loads values from a touchstone file.

The work of this function is done through the `touchstone` class.

**Parameters** `filename` : str or file-object

touchstone file name.

**Notes**

only the scattering parameters format is supported at the moment

**skrf.network.Network.renumber**

`Network.renumber` (*from\_ports, to\_ports*)

renumbers some ports of a two port Network

**Parameters** `from_ports` : list-like

`to_ports`: list-like :

**Examples**

To flip the ports of a 2-port network 'foo': `>>> foo.renumber( [0,1], [1,0] )`

To rotate the ports of a 3-port network 'bar' so that port 0 becomes port 1: `>>> bar.renumber( [0,1,2], [1,2,0] )`

To swap the first and last ports of a network 'duck': `>>> duck.renumber( [0,-1], [-1,0] )`

**skrf.network.Network.resample**

`Network.resample` (*npoints, \*\*kwargs*)

Interpolate network based on a new number of frequency points

**Parameters** `npoints` : int

number of frequency points

**\*\*kwargs** : keyword arguments

passed to `scipy.interpolate.interpld()` initializer.

**See Also:**

**`interpolate_self`** same functionality but takes a Frequency object

**`interpolate`** same functionality but takes a Frequency object and returns a new Network, instead of updating itself.

**Notes**

The function `resample()` is an alias for `interpolate_self_npoints()`.

### Examples

```
In [2]: n = rf.data.ring_slot
```

```
In [3]: n
```

```
Out[3]: 2-Port Network: 'ring slot', 75-110 GHz, 501 pts, z0=[ 50.+0.j 50.+0.j]
```

```
In [4]: n.resample(501) # resample is an alias
```

```
In [5]: n
```

```
Out[5]: 2-Port Network: 'ring slot', 75-110 GHz, 501 pts, z0=[ 50.+0.j 50.+0.j]
```

### skrf.network.Network.write

`Network.write` (*file=None, \*args, \*\*kwargs*)

Write the Network to disk using the `pickle` module.

The resultant file can be read either by using the Networks constructor, `__init__()`, the read method `read()`, or the general read function `skrf.io.general.read()`

**Parameters** `file` : str or file-object

filename or a file-object. If left as `None` then the filename will be set to `Network.name`, if its not `None`. If both are `None`, `ValueError` is raised.

**\*args, \*\*kwargs** : :

passed through to `write()`

**See Also:**

`skrf.io.general.write` write any skrf object

`skrf.io.general.read` read any skrf object

### Notes

If the `self.name` is not `None` and `file` is can left as `None` and the resultant file will have the `.ntwk` extension appended to the filename.

### Examples

```
>>> n = rf.N(f=[1,2,3],s=[1,1,1],z0=50, name = 'open')
```

```
>>> n.write()
```

```
>>> n2 = rf.read('open.ntwk')
```

### skrf.network.Network.write\_touchstone

`Network.write_touchstone` (*filename=None, dir='./, write\_z0=False*)

write a contents of the `Network` to a touchstone file.

**Parameters** `filename` : a string, optional

touchstone filename, without extension. if 'None', then will use the network's name.



**dir** : string, optional

the directory to save the file in. Defaults to cwd `'.'`.

**write\_z0** : boolean

write impedance information into touchstone as comments, like Ansoft HFSS does

### Notes

format supported at the moment is, HZ S RI

The functionality of this function should take place in the `touchstone` class.

## 3.2.2 Connecting Networks

<code>connect(ntwkA, k, ntwkB, l[, num])</code>	connect two n-port networks together.
<code>innerconnect(ntwkA, k, l[, num])</code>	connect ports of a single n-port network.
<code>cascade(ntwkA, ntwkB)</code>	Cascade two 2-port Networks together
<code>de_embed(ntwkA, ntwkB)</code>	De-embed <i>ntwkA</i> from <i>ntwkB</i> .
<code>flip(a)</code>	invert the ports of a networks s-matrix, 'flipping' it over

### skrf.network.connect

`skrf.network.connect(ntwkA, k, ntwkB, l, num=1)`

connect two n-port networks together.

specifically, connect ports  $k$  thru  $k+num-1$  on *ntwkA* to ports  $l$  thru  $l+num-1$  on *ntwkB*. The resultant network has  $(ntwkA.nports+ntwkB.nports-2*num)$  ports. The port indices ('k','l') start from 0. Port impedances **are** taken into account.

**Parameters** `ntwkA` : *Network*

network 'A'

**k** : int

starting port index on *ntwkA* ( port indices start from 0 )

**ntwkB** : *Network*

network 'B'

**l** : int

starting port index on *ntwkB*

**num** : int

number of consecutive ports to connect (default 1)

**Returns** `ntwkC` : *Network*

new network of rank  $(ntwkA.nports + ntwkB.nports - 2*num)$

**See Also:**

`connect_s` actual S-parameter connection algorithm.

`innerconnect_s` actual S-parameter connection algorithm.

## Notes

the effect of mis-matched port impedances is handled by inserting a 2-port ‘mismatch’ network between the two connected ports. This mismatch Network is calculated with the `impedance_mismatch()` function.

## Examples

To implement a *cascade* of two networks

```
>>> ntwkA = rf.Network('ntwkA.s2p')
>>> ntwkB = rf.Network('ntwkB.s2p')
>>> ntwkC = rf.connect(ntwkA, 1, ntwkB, 0)
```

## skrf.network.innerconnect

`skrf.network.innerconnect` (*ntwkA*, *k*, *l*, *num=1*)

connect ports of a single n-port network.

this results in a (n-2)-port network. remember port indices start from 0.

**Parameters** `ntwkA`: *Network*

network ‘A’

**k,l**: int

starting port indices on ntwkA ( port indices start from 0 )

**num**: int

number of consecutive ports to connect

**Returns** `ntwkC`: *Network*

new network of rank (ntwkA.nports - 2\*num)

### See Also:

`connect_s` actual S-parameter connection algorithm.

`innerconnect_s` actual S-parameter connection algorithm.

## Notes

a 2-port ‘mismatch’ network is inserted between the connected ports if their impedances are not equal.

## Examples

To connect ports ‘0’ and port ‘1’ on ntwkA

```
>>> ntwkA = rf.Network('ntwkA.s3p')
>>> ntwkC = rf.innerconnect(ntwkA, 0, 1)
```

### skrf.network.cascade

`skrf.network.cascade` (*ntwkA*, *ntwkB*)

Cascade two 2-port Networks together

Connects port 1 of *ntwkA* to port 0 of *ntwkB*. This calls `connect(ntwkA,1, ntwkB,0)`, which is a more general function.

**Parameters** *ntwkA* : `Network`

network *ntwkA*

**ntwkB** : `Network`

network *ntwkB*

**Returns** *C* : `Network`

the resultant network of *ntwkA* cascaded with *ntwkB*

**See Also:**

`connect` connects two Networks together at arbitrary ports.

### skrf.network.de\_embed

`skrf.network.de_embed` (*ntwkA*, *ntwkB*)

De-embed *ntwkA* from *ntwkB*.

This calls `ntwkA.inv ** ntwkB`. The syntax of cascading an inverse is more explicit, it is recommended that it be used instead of this function.

**Parameters** *ntwkA* : `Network`

network *ntwkA*

**ntwkB** : `Network`

network *ntwkB*

**Returns** *C* : `Network`

the resultant network of *ntwkB* de-embedded from *ntwkA*

**See Also:**

`connect` connects two Networks together at arbitrary ports.

### skrf.network.flip

`skrf.network.flip` (*a*)

invert the ports of a networks s-matrix, ‘flipping’ it over

**Parameters** *a* : `numpy.ndarray`

scattering parameter matrix. shape should be should be 2x2, or fx2x2

**Returns** *a'* : `numpy.ndarray`

flipped scattering parameter matrix, ie interchange of port 0 and port 1

### 3.2.3 Interpolation and Stitching

<code>Network.resample(npoints, **kwargs)</code>	Interpolate network based on a new number of frequency points
<code>Network.interpolate(new_frequency, **kwargs)</code>	Return an interpolated network, from a new :class:`~skrf.frequency`
<code>Network.interpolate_self(new_frequency, **kwargs)</code>	Interpolates s-parameters given a new
<code>Network.interpolate_from_f(f[, interp_kwargs])</code>	Interpolates s-parameters from a frequency vector.
<code>stitch(ntwkA, ntwkB, **kwargs)</code>	Stitches ntwkA and ntwkB together.

#### skrf.network.Network.resample

`Network.resample` (*npoints*, *\*\*kwargs*)

Interpolate network based on a new number of frequency points

**Parameters** `npoints` : int

number of frequency points

**\*\*kwargs** : keyword arguments

passed to `scipy.interpolate.interp1d()` initializer.

**See Also:**

`interpolate_self` same functionality but takes a Frequency object

`interpolate` same functionality but takes a Frequency object and returns a new Network, instead of updating itself.

**Notes**

The function `resample()` is an alias for `interpolate_self_npoints()`.

**Examples**

```
In [2]: n = rf.data.ring_slot
```

```
In [3]: n
```

```
Out[3]: 2-Port Network: 'ring slot', 75-110 GHz, 501 pts, z0=[ 50.+0.j 50.+0.j]
```

```
In [4]: n.resample(501) # resample is an alias
```

```
In [5]: n
```

```
Out[5]: 2-Port Network: 'ring slot', 75-110 GHz, 501 pts, z0=[ 50.+0.j 50.+0.j]
```

#### skrf.network.Network.interpolate

`Network.interpolate` (*new\_frequency*, *\*\*kwargs*)

Return an interpolated network, from a new :class:`~skrf.frequency.Frequency`.

Interpolate the networks s-parameters linearly in real and imaginary components. Other interpolation types can be used by passing appropriate *\*\*kwargs*. This function *returns* an interpolated Network. Alternatively `interpolate_self()` will interpolate self.

**Parameters** `new_frequency` : `Frequency`

frequency information to interpolate

**\*\*kwargs** : keyword arguments

passed to `scipy.interpolate.interp1d()` initializer.

**Returns** **result** : `Network`

an interpolated `Network`

**See Also:**

`resample`, `interpolate_self`, `interpolate_from_f`

## Notes

See `scipy.interpolate.interpolate.interp1d()` for useful kwargs. For example

**kind** [str or int] Specifies the kind of interpolation as a string ('linear', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic') or as an integer specifying the order of the spline interpolator to use.

## Examples

```
In [2]: n = rf.data.ring_slot
```

```
In [3]: n
```

```
Out[3]: 2-Port Network: 'ring slot', 75-110 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j]
```

```
In [4]: new_freq = rf.Frequency(75,110,501,'ghz')
```

```
In [5]: n.interpolate(new_freq, kind = 'cubic')
```

```
Out[5]: 2-Port Network: 'ring slot', 75-110 GHz, 501 pts, z0=[ 50.+0.j 50.+0.j]
```

## `skrf.network.Network.interpolate_self`

`Network.interpolate_self` (*new\_frequency*, **\*\*kwargs**)

Interpolates s-parameters given a new :class:`~skrf.frequency.Frequency` object.

See `interpolate()` for more information.

**Parameters** **new\_frequency** : `Frequency`

frequency information to interpolate at

**\*\*kwargs** : keyword arguments

passed to `scipy.interpolate.interp1d()` initializer.

**See Also:**

`resample`, `interpolate`, `interpolate_from_f`

## `skrf.network.Network.interpolate_from_f`

`Network.interpolate_from_f` (*f*, *interp\_kwargs*={}, **\*\*kwargs**)

Interpolates s-parameters from a frequency vector.

Given a frequency vector, and optionally a *unit* (see **\*\*kwargs**) , interpolate the networks s-parameters linearly in real and imaginary components.

See `interpolate()` for more information.

**Parameters** `new_frequency`: `Frequency`

frequency information to interpolate at

**interp\_kwargs** :

dictionary of kwargs to be passed through to `scipy.interpolate.interpdl()`

**\*\*kwargs** :

passed to `scipy.interpolate.interpdl()` initializer.

**See Also:**

`resample`, `interpolate`, `interpolate_self`

### Notes

This creates a new `Frequency` object using the method `from_f()`, and then calls `interpolate_self()`.

## skrf.network.stitch

`skrf.network.stitch(ntwkA, ntwkB, **kwargs)`

Stitches `ntwkA` and `ntwkB` together.

Concatenates two networks' data. Given two networks that cover different frequency bands this can be used to combine their data into a single network.

**Parameters** `ntwkA`, `ntwkB`: `Network` objects

Networks to stitch together

**\*\*kwargs**: keyword args

passed to `Network` constructor, for output network

**Returns** `ntwkC`: `Network`

result of stitching the networks `ntwkA` and `ntwkB` together

### Examples

```
>>> from skrf.data import wr2p2_line, wr1p5_line
>>> rf.stitch(wr2p2_line, wr1p5_line)
2-Port Network: 'wr2p2,line', 330-750 GHz, 402 pts, z0=[ 50.+0.j 50.+0.j]
```

## 3.2.4 IO

<code>skrf.io.general.read(file, *args, **kwargs)</code>	Read skrf object[s] from a pickle file
<code>skrf.io.general.write(file, obj[, overwrite])</code>	Write skrf object[s] to a file
<code>Network.write([file])</code>	Write the <code>Network</code> to disk using the <code>pickle</code> module.
<code>Network.write_touchstone([filename, dir, ...])</code>	write a contents of the <code>Network</code> to a touchstone file.
Continued on next page	

Table 3.9 – continued from previous page

<code>Network.read(*args, **kwargs)</code>	Read a Network from a 'ntwk' file
--	-----------------------------------

### 3.2.5 Noise

<code>Network.add_noise_polar(mag_dev, phase_dev, ...)</code>	adds a complex zero-mean gaussian white-noise.
<code>Network.add_noise_polar_flatband(mag_dev, ...)</code>	adds a flatband complex zero-mean gaussian white-noise signal of
<code>Network.multiply_noise(mag_dev, phase_dev, ...)</code>	multiplies a complex bivariate gaussian white-noise signal

#### skrf.network.Network.add\_noise\_polar

`Network.add_noise_polar` (*mag\_dev*, *phase\_dev*, *\*\*kwargs*)

adds a complex zero-mean gaussian white-noise.

adds a complex zero-mean gaussian white-noise of a given standard deviation for magnitude and phase

**Parameters** `mag_dev` : number

standard deviation of magnitude

`phase_dev` : number

standard deviation of phase [in degrees]

#### skrf.network.Network.add\_noise\_polar\_flatband

`Network.add_noise_polar_flatband` (*mag\_dev*, *phase\_dev*, *\*\*kwargs*)

adds a flatband complex zero-mean gaussian white-noise signal of given standard deviations for magnitude and phase

**Parameters** `mag_dev` : number

standard deviation of magnitude

`phase_dev` : number

standard deviation of phase [in degrees]

#### skrf.network.Network.multiply\_noise

`Network.multiply_noise` (*mag\_dev*, *phase\_dev*, *\*\*kwargs*)

multiplies a complex bivariate gaussian white-noise signal of given standard deviations for magnitude and phase. magnitude mean is 1, phase mean is 0

**takes:** `mag_dev`: standard deviation of magnitude `phase_dev`: standard deviation of phase [in degrees] `n_ports`: number of ports. default to 1

**returns:** nothing

### 3.2.6 Supporting Functions

<code>inv(s)</code>	Calculates 'inverse' s-parameter matrix, used for de-embedding
<code>connect_s(A, k, B, l)</code>	connect two n-port networks' s-matrices together.
Continued on next page	

Table 3.11 – continued from previous page

<code>innerconnect_s(A, k, l)</code>	connect two ports of a single n-port network's s-matrix.
<code>s2z(s[, z0])</code>	Convert scattering parameters [#]_ to impedance parameters [#]_ ..
<code>s2y(s[, z0])</code>	convert scattering parameters [#]_ to admittance parameters [#]_
<code>s2t(s)</code>	Converts scattering parameters [#]_ to scattering transfer parameters [#]_ .
<code>z2s(z[, z0])</code>	convert impedance parameters [#]_ to scattering parameters [#]_
<code>z2y(z)</code>	convert impedance parameters [#]_ to admittance parameters [#]_
<code>z2t(z)</code>	Not Implemented yet
<code>y2s(y[, z0])</code>	convert admittance parameters [#]_ to scattering parameters [#]_
<code>y2z(y)</code>	convert admittance parameters [#]_ to impedance parameters [#]_
<code>y2t(y)</code>	Not Implemented Yet
<code>t2s(t)</code>	converts scattering transfer parameters [#]_ to scattering parameters [#]_
<code>t2z(t)</code>	Not Implemented Yet
<code>t2y(t)</code>	Not Implemented Yet

### skrf.network.inv

`skrf.network.inv(s)`

Calculates ‘inverse’ s-parameter matrix, used for de-embedding

This is not literally the inverse of the s-parameter matrix. Instead, it is defined such that the inverse of the s-matrix cascaded with itself is unity.

$$inv(s) = t2s(s2t(s)^{-1})$$

where  $x^{-1}$  is the matrix inverse. In words, this is the inverse of the scattering transfer parameters matrix transformed into a scattering parameters matrix.

**Parameters** `s` : `numpy.ndarray` (shape fx2x2)  
scattering parameter matrix.

**Returns** `s'` : `numpy.ndarray`  
inverse scattering parameter matrix.

**See Also:**

**t2s** converts scattering transfer parameters to scattering parameters

**s2t** converts scattering parameters to scattering transfer parameters

### skrf.network.connect\_s

`skrf.network.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.

**Parameters** `A` : `numpy.ndarray`  
S-parameter matrix of *A*, shape is `fxn`  
`k` : `int`  
port index on *A* (port indices start from 0)



**B** : `numpy.ndarray`

S-parameter matrix of  $B$ , shape is  $fxn \times nxn$

**l** : `int`

port index on  $B$

**Returns** **C** : `numpy.ndarray`

new S-parameter matrix

#### See Also:

`connect` operates on `Network` types

`innerconnect_s` function which implements the connection connection algorithm

#### Notes

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

### skrf.network.innerconnect\_s

`skrf.network.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.

**Parameters** **A** : `numpy.ndarray`

S-parameter matrix of  $A$ , shape is  $fxn \times nxn$

**k** : `int`

port index on  $A$  (port indices start from 0)

**l** : `int`

port index on  $A$

**Returns** **C** : `numpy.ndarray`

new S-parameter matrix

#### Notes

The algorithm used to calculate the resultant network is called a 'sub-network growth', can be found in <sup>6</sup>. The original paper describing the algorithm is given in <sup>7</sup>.

<sup>6</sup> Compton, R.C.; , "Perspectives in microwave circuit analysis," Circuits and Systems, 1989., Proceedings of the 32nd Midwest Symposium on , vol., no., pp.716-718 vol.2, 14-16 Aug 1989. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=101955&isnumber=3167>

<sup>7</sup> Filipsson, Gunnar; , "A New General Computer Algorithm for S-Matrix Calculation of Interconnected Multiports," Microwave Conference, 1981. 11th European , vol., no., pp.700-704, 7-11 Sept. 1981. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4131699&isnumber=4131585>

## References

### skrf.network.s2z

skrf.network.**s2z** (*s*, *z0*=50)

Convert scattering parameters<sup>8</sup> to impedance parameters<sup>9</sup>

$$z = \sqrt{z_0} \cdot (I + s)(I - s)^{-1} \cdot \sqrt{z_0}$$

**Parameters** *s* : complex array-like

scattering parameters

**z0** : complex array-like or number

port impedances

**Returns** *z* : complex array-like

impedance parameters

**See Also:**

`s2z`, `s2y`, `s2t`, `z2s`, `z2y`, `z2t`, `y2s`, `y2z`, `y2t`, `t2s`, `t2z`, `t2y`, `Network.s`, `Network.y`, `Network.z`, `Network.t`

## References

### skrf.network.s2y

skrf.network.**s2y** (*s*, *z0*=50)

convert scattering parameters<sup>10</sup> to admittance parameters<sup>11</sup>

$$y = \sqrt{y_0} \cdot (I - s)(I + s)^{-1} \cdot \sqrt{y_0}$$

**Parameters** *s* : complex array-like

scattering parameters

**z0** : complex array-like or number

port impedances

**Returns** *y* : complex array-like

admittance parameters

**See Also:**

`s2z`, `s2y`, `s2t`, `z2s`, `z2y`, `z2t`, `y2s`, `y2z`, `y2t`, `t2s`, `t2z`, `t2y`, `Network.s`, `Network.y`, `Network.z`, `Network.t`

---

<sup>8</sup> <http://en.wikipedia.org/wiki/S-parameters>

<sup>9</sup> [http://en.wikipedia.org/wiki/impedance\\_parameters](http://en.wikipedia.org/wiki/impedance_parameters)

<sup>10</sup> <http://en.wikipedia.org/wiki/S-parameters>

<sup>11</sup> [http://en.wikipedia.org/wiki/Admittance\\_parameters](http://en.wikipedia.org/wiki/Admittance_parameters)

## References

### skrf.network.s2t

`skrf.network.s2t` (*s*)

Converts scattering parameters <sup>12</sup> to scattering transfer parameters <sup>13</sup>.

transfer parameters are also referred to as ‘wave cascading matrix’, this function only operates on 2-port networks.

**Parameters** *s* : `numpy.ndarray` (shape `fx2x2`)

scattering parameter matrix

**Returns** *t* : `numpy.ndarray`

scattering transfer parameters (aka wave cascading matrix)

#### See Also:

`inv` calculates inverse s-parameters

`s2z`, `s2y`, `s2t`, `z2s`, `z2y`, `z2t`, `y2s`, `y2z`, `y2z`, `t2s`, `t2z`, `t2y`, `Network.s`, `Network.y`, `Network.z`, `Network.t`

## References

### skrf.network.z2s

`skrf.network.z2s` (*z*, *z0=50*)

convert impedance parameters <sup>14</sup> to scattering parameters <sup>15</sup>

$$s = (\sqrt{y_0} \cdot z \cdot \sqrt{y_0} - I)(\sqrt{y_0} \cdot z \cdot \sqrt{y_0} + I)^{-1}$$

**Parameters** *z* : complex array-like

impedance parameters

*z0* : complex array-like or number

port impedances

**Returns** *s* : complex array-like

scattering parameters

#### See Also:

`s2z`, `s2y`, `s2t`, `z2s`, `z2y`, `z2t`, `y2s`, `y2z`, `y2z`, `t2s`, `t2z`, `t2y`, `Network.s`, `Network.y`, `Network.z`, `Network.t`

<sup>12</sup> <http://en.wikipedia.org/wiki/S-parameters>

<sup>13</sup> [http://en.wikipedia.org/wiki/Scattering\\_transfer\\_parameters#Scattering\\_transfer\\_parameters](http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters)

<sup>14</sup> [http://en.wikipedia.org/wiki/impedance\\_parameters](http://en.wikipedia.org/wiki/impedance_parameters)

<sup>15</sup> <http://en.wikipedia.org/wiki/S-parameters>

## References

### skrf.network.z2y

skrf.network.**z2y**(z)  
convert impedance parameters <sup>16</sup> to admittance parameters <sup>17</sup>

$$y = z^{-1}$$

**Parameters** z : complex array-like

impedance parameters

**Returns** y : complex array-like

admittance parameters

#### See Also:

s2z, s2y, s2t, z2s, z2y, z2t, y2s, y2z, y2z, t2s, t2z, t2y, Network.s, Network.y, Network.z, Network.t

## References

### skrf.network.z2t

skrf.network.**z2t**(z)  
Not Implemented yet  
convert impedance parameters <sup>18</sup> to scattering transfer parameters <sup>19</sup>

**Parameters** z : complex array-like or number

impedance parameters

**Returns** s : complex array-like or number

scattering parameters

#### See Also:

s2z, s2y, s2t, z2s, z2y, z2t, y2s, y2z, y2z, t2s, t2z, t2y, Network.s, Network.y, Network.z, Network.t

---

<sup>16</sup> [http://en.wikipedia.org/wiki/impedance\\_parameters](http://en.wikipedia.org/wiki/impedance_parameters)

<sup>17</sup> [http://en.wikipedia.org/wiki/Admittance\\_parameters](http://en.wikipedia.org/wiki/Admittance_parameters)

<sup>18</sup> [http://en.wikipedia.org/wiki/impedance\\_parameters](http://en.wikipedia.org/wiki/impedance_parameters)

<sup>19</sup> [http://en.wikipedia.org/wiki/Scattering\\_transfer\\_parameters#Scattering\\_transfer\\_parameters](http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters)

## References

### skrf.network.y2s

skrf.network.**y2s** (*y*, *z0=50*)  
 convert admittance parameters <sup>20</sup> to scattering parameters <sup>21</sup>

$$s = (I - \sqrt{z_0} \cdot y \cdot \sqrt{z_0})(I + \sqrt{z_0} \cdot y \cdot \sqrt{z_0})^{-1}$$

**Parameters** *y* : complex array-like

admittance parameters

**z0** : complex array-like or number

port impedances

**Returns** *s* : complex array-like or number

scattering parameters

**See Also:**

*s2z*, *s2y*, *s2t*, *z2s*, *z2y*, *z2t*, *y2s*, *y2z*, *y2z*, *t2s*, *t2z*, *t2y*, *Network.s*, *Network.y*, *Network.z*, *Network.t*

## References

### skrf.network.y2z

skrf.network.**y2z** (*y*)  
 convert admittance parameters <sup>22</sup> to impedance parameters <sup>23</sup>

$$z = y^{-1}$$

**Parameters** *y* : complex array-like

admittance parameters

**Returns** *z* : complex array-like

impedance parameters

**See Also:**

*s2z*, *s2y*, *s2t*, *z2s*, *z2y*, *z2t*, *y2s*, *y2z*, *y2z*, *t2s*, *t2z*, *t2y*, *Network.s*, *Network.y*, *Network.z*, *Network.t*

<sup>20</sup> [http://en.wikipedia.org/wiki/Admittance\\_parameters](http://en.wikipedia.org/wiki/Admittance_parameters)

<sup>21</sup> <http://en.wikipedia.org/wiki/S-parameters>

<sup>22</sup> [http://en.wikipedia.org/wiki/Admittance\\_parameters](http://en.wikipedia.org/wiki/Admittance_parameters)

<sup>23</sup> [http://en.wikipedia.org/wiki/impedance\\_parameters](http://en.wikipedia.org/wiki/impedance_parameters)

## References

### skrf.network.y2t

skrf.network.y2t(y)

Not Implemented Yet

convert admittance parameters <sup>24</sup> to scattering-transfer parameters <sup>25</sup>

**Parameters** y : complex array-like or number

impedance parameters

**Returns** t : complex array-like or number

scattering parameters

#### See Also:

s2z, s2y, s2t, z2s, z2y, z2t, y2s, y2z, y2z, t2s, t2z, t2y, Network.s, Network.y, Network.z, Network.t

## References

### skrf.network.t2s

skrf.network.t2s(t)

converts scattering transfer parameters <sup>26</sup> to scattering parameters <sup>27</sup>

transfer parameters are also referred to as ‘wave cascading matrix’, this function only operates on 2-port networks. this function only operates on 2-port scattering parameters.

**Parameters** t : numpy.ndarray (shape fx2x2)

scattering transfer parameters

**Returns** s : numpy.ndarray

scattering parameter matrix.

#### See Also:

**inv** calculates inverse s-parameters

s2z, s2y, s2t, z2s, z2y, z2t, y2s, y2z, y2z, t2s, t2z, t2y, Network.s, Network.y, Network.z, Network.t

## References

### skrf.network.t2z

skrf.network.t2z(t)

Not Implemented Yet

---

<sup>24</sup> [http://en.wikipedia.org/wiki/Admittance\\_parameters](http://en.wikipedia.org/wiki/Admittance_parameters)

<sup>25</sup> [http://en.wikipedia.org/wiki/Scattering\\_transfer\\_parameters#Scattering\\_transfer\\_parameters](http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters)

<sup>26</sup> [http://en.wikipedia.org/wiki/Scattering\\_transfer\\_parameters#Scattering\\_transfer\\_parameters](http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters)

<sup>27</sup> <http://en.wikipedia.org/wiki/S-parameters>

Convert scattering transfer parameters <sup>28</sup> to impedance parameters <sup>29</sup>

**Parameters** `t` : complex array-like or number  
impedance parameters

**Returns** `z` : complex array-like or number  
scattering parameters

**See Also:**

`s2z`, `s2y`, `s2t`, `z2s`, `z2y`, `z2t`, `y2s`, `y2z`, `y2z`, `t2s`, `t2z`, `t2y`, `Network.s`, `Network.y`, `Network.z`, `Network.t`

**References**

### skrf.network.t2y

`skrf.network.t2y(t)`  
Not Implemented Yet

Convert scattering transfer parameters to admittance parameters <sup>30</sup>

**Parameters** `t` : complex array-like or number  
t-parameters

**Returns** `y` : complex array-like or number  
admittance parameters

**See Also:**

`s2z`, `s2y`, `s2t`, `z2s`, `z2y`, `z2t`, `y2s`, `y2z`, `y2z`, `t2s`, `t2z`, `t2y`, `Network.s`, `Network.y`, `Network.z`, `Network.t`

**References**

## 3.2.7 Misc Functions

---

<code>average(list_of_networks)</code>	Calculates the average network from a list of Networks.
<code>Network.nudge([amount])</code>	Perturb s-parameters by small amount.

---

### skrf.network.average

`skrf.network.average(list_of_networks)`  
Calculates the average network from a list of Networks.

This is complex average of the s-parameters for a list of Networks.

**Parameters** `list_of_networks` : list of `Network` objects  
the list of networks to average

**Returns** `ntwk` : `Network`

<sup>28</sup> [http://en.wikipedia.org/wiki/Scattering\\_transfer\\_parameters#Scattering\\_transfer\\_parameters](http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters)

<sup>29</sup> [http://en.wikipedia.org/wiki/impedance\\_parameters](http://en.wikipedia.org/wiki/impedance_parameters)

<sup>30</sup> [http://en.wikipedia.org/wiki/Scattering\\_transfer\\_parameters#Scattering\\_transfer\\_parameters](http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters)

the resultant averaged Network

#### Notes

This same function can be accomplished with properties of a `NetworkSet` class.

#### Examples

```
>>> ntwk_list = [rf.Network('myntwk.slp'), rf.Network('myntwk2.slp')]
>>> mean_ntwk = rf.average(ntwk_list)
```

### `skrf.network.Network.nudge`

`Network.nudge` (*amount=1e-12*)

Perturb s-parameters by small amount.

This is useful to work-around numerical bugs.

**Parameters** `amount`: number,  
amount to add to s parameters

#### Notes

This function is `self.s = self.s + 1e-12`

## 3.3 networkSet (`skrf.networkSet`)

Provides a class representing an un-ordered set of n-port microwave networks.

Frequently one needs to make calculations, such as mean or standard deviation, on an entire set of n-port networks. To facilitate these calculations the `NetworkSet` class provides convenient ways to make such calculations.

The results are returned in `Network` objects, so they can be plotted and saved in the same way one would do with a `Network`.

The functionality in this module is provided as methods and properties of the `NetworkSet` Class.

### 3.3.1 NetworkSet Class

---

`NetworkSet(ntwk_set[, name])` A set of Networks.

---

#### `skrf.networkSet.NetworkSet`

**class** `skrf.networkSet.NetworkSet` (*ntwk\_set, name=None*)  
A set of Networks.

This class allows functions on sets of Networks, such as mean or standard deviation, to be calculated conveniently. The results are returned in `Network` objects, so that they may be plotted and saved in like `Network`



objects.

This class also provides methods which can be used to plot uncertainty bounds for a set of `Network`.

The names of the `NetworkSet` properties are generated dynamically upon initialization, and thus documentation for individual properties and methods is not available. However, the properties do follow the convention:

```
>>> my_network_set.function_name_network_property_name
```

For example, the complex average (mean) `Network` for a `NetworkSet` is:

```
>>> my_network_set.mean_s
```

This accesses the property 's', for each element in the set, and **then** calculates the 'mean' of the resultant set. The order of operations is important.

Results are returned as `Network` objects, so they may be plotted or saved in the same way as for `Network` objects:

```
>>> my_network_set.mean_s.plot_s_mag()
>>> my_network_set.mean_s.write_touchstone('mean_response')
```

If you are calculating functions that return scalar variables, then the result is accessible through the `Network` property `.s_re`. For example:

```
>>> std_s_deg = my_network_set.std_s_deg
```

This result would be plotted by:

```
>>> std_s_deg.plot_s_re()
```

The operators, properties, and methods of `NetworkSet` object are dynamically generated by private methods

- `__add_a_operator()`
- `__add_a_func_on_property()`
- `__add_a_element_wise_method()`
- `__add_a_plot_uncertainty()`

thus, documentation on the individual methods and properties are not available.

## Attributes

<code>inv</code>	
<code>mean_s_db</code>	the mean magnitude in dB.
<code>std_s_db</code>	the mean magnitude in dB.

### `skrf.networkSet.NetworkSet.inv`

`NetworkSet.inv`

### `skrf.networkSet.NetworkSet.mean_s_db`

`NetworkSet.mean_s_db`  
the mean magnitude in dB.

**note:**

**the mean is taken on the magnitude before converted to db, so `magnitude_2_db( mean(s_mag))` which is NOT the same as `mean(s_db)`**

**skrf.networkSet.NetworkSet.std\_s\_db**

`NetworkSet.std_s_db`  
the mean magnitude in dB.

**note:**

**the mean is taken on the magnitude before converted to db, so `magnitude_2_db( mean(s_mag))` which is NOT the same as `mean(s_db)`**

**Methods**

<code>__init__</code>	Initializer for NetworkSet
<code>copy</code>	copies each network of the network set.
<code>element_wise_method</code>	calls a given method of each element and returns the result as
<code>from_zip</code>	creates a NetworkSet from a zipfile of touchstones.
<code>plot_logsigma</code>	plots the uncertainty for the set in units of log-sigma.
<code>plot_uncertainty_bounds_component</code>	plots mean value of the NetworkSet with +- uncertainty bounds
<code>plot_uncertainty_bounds_s</code>	Plots complex uncertainty bounds plot on smith chart.
<code>plot_uncertainty_bounds_s_db</code>	this just calls
<code>plot_uncertainty_decomposition</code>	plots the total and component-wise uncertainty
<code>set_wise_function</code>	calls a function on a specific property of the networks in
<code>signature</code>	visualization of relative changes in a NetworkSet.
<code>uncertainty_ntwk_triplet</code>	returns a 3-tuple of Network objects which contain the
<code>write</code>	Write the NetworkSet to disk using <code>write()</code>

**skrf.networkSet.NetworkSet.\_\_init\_\_**

`NetworkSet.__init__(ntwk_set, name=None)`  
Initializer for NetworkSet

**Parameters** `ntwk_set`: list of `Network` objects

the set of `Network` objects

**name**: string

the name of the NetworkSet, given to the Networks returned from properties of this class.

**skrf.networkSet.NetworkSet.copy**

`NetworkSet.copy()`  
copies each network of the network set.

**skrf.networkSet.NetworkSet.element\_wise\_method**

`NetworkSet.element_wise_method` (*network\_method\_name*, \*args, \*\*kwargs)  
 calls a given method of each element and returns the result as a new NetworkSet if the output is a Network.

**skrf.networkSet.NetworkSet.from\_zip**

**classmethod** `NetworkSet.from_zip` (*zip\_file\_name*, *sort\_filenames=True*, \*args, \*\*kwargs)  
 creates a NetworkSet from a zipfile of touchstones.

**Parameters** `zip_file_name` : string

name of zipfile

**sort\_filenames: Boolean :**

sort the filenames in teh zip file before constructing the NetworkSet

**\*args,\*\*kwargs :** arguments

passed to NetworkSet constructor

**Examples**

```
>>> import skrf as rf
>>> my_set = rf.NetworkSet.from_zip('myzip.zip')
```

**skrf.networkSet.NetworkSet.plot\_logsigma**

`NetworkSet.plot_logsigma` (*label\_axis=True*, \*args, \*\*kwargs)  
 plots the uncertainty for the set in units of log-sigma. Log-sigma is the complex standard deviation, plotted in units of dB's.

**Parameters** \*args, \*\*kwargs : arguments

passed to self.std\_s.plot\_s\_db()

**skrf.networkSet.NetworkSet.plot\_uncertainty\_bounds\_component**

`NetworkSet.plot_uncertainty_bounds_component` (*attribute*, *m=0*, *n=0*, *type='shade'*,  
*n\_deviations=3*, *alpha=0.3*,  
*color\_error=None*, *markevery\_error=20*,  
*ax=None*, *ppf=None*, *kwargs\_error={}*,  
 \*args, \*\*kwargs)

plots mean value of the NetworkSet with +- uncertainty bounds in an Network's attribute. This is designed to represent uncertainty in a scalar component of the s-parameter. for example plotting the uncertainty in the magnitude would be expressed by,

$\text{mean}(\text{abs}(s)) \pm \text{std}(\text{abs}(s))$

the order of mean and abs is important.

**takes:** attribute: attribute of Network type to analyze [string] m: first index of attribute matrix [int] n: second index of attribute matrix [int] type: ['shade' | 'bar'], type of plot to draw n\_deviations: number of std deviations to plot as bounds [number] alpha: passed to matplotlib.fill\_between() command. [number, 0-1] color\_error: color of the +- std dev fill shading markevery\_error: if type=='bar', this controls frequency

of error bars

ax: Axes to plot on  
ppf: post processing function. a function applied to the upper and low

**\*args,\*\*kwargs:** passed to `Network.plot_s_re` command used to plot mean response

**kwargs\_error:** dictionary of kwargs to pass to the `fill_between` or errorbar plot command depending on value of type.

**returns:** None

**Note:** for phase uncertainty you probably want `s_deg_unwrap`, or similar. uncertainty for wrapped phase blows up at  $\pm\pi$ .

### `skrf.networkSet.NetworkSet.plot_uncertainty_bounds_s`

`NetworkSet.plot_uncertainty_bounds_s` (*multiplier=200, \*args, \*\*kwargs*)

Plots complex uncertainty bounds plot on smith chart.

This function plots the complex uncertainty of a `NetworkSet` as circles on the smith chart. At each frequency a circle with radii proportional to the complex standard deviation of the set at that frequency is drawn. Due to the fact that the `markersize` argument is in pixels, the radii can scaled by the input argument *multiplier*.

**default kwargs are** { 'marker':'o', 'color':'b', 'mew':0, 'ls':'-', 'alpha':.1, 'label':None, }

**Parameters** `multiplier` : float

controls the circle sizes, by multiples of the standard deviation.

### `skrf.networkSet.NetworkSet.plot_uncertainty_bounds_s_db`

`NetworkSet.plot_uncertainty_bounds_s_db` (*\*args, \*\*kwargs*)

**this just calls** `plot_uncertainty_bounds(attribute='s_mag',ppf:mf.magnitude_2_db*args,**kwargs)`

see `plot_uncertainty_bounds` for help

### `skrf.networkSet.NetworkSet.plot_uncertainty_decomposition`

`NetworkSet.plot_uncertainty_decomposition` (*m=0, n=0*)

plots the total and component-wise uncertainty

**Parameters** `m` : int

first s-parameters index

**n** :

second s-parameter index

**skrf.networkSet.NetworkSet.set\_wise\_function**

`NetworkSet.set_wise_function` (*func, a\_property, \*args, \*\*kwargs*)  
 calls a function on a specific property of the networks in this NetworkSet.

**example:** `my_ntwk_set.set_wise_func(mean,'s')`

**skrf.networkSet.NetworkSet.signature**

`NetworkSet.signature` (*m=0, n=0, from\_mean=False, operation='\_\_sub\_\_', component='s\_mag', vmax=None, \*args, \*\*kwargs*)  
 visualization of relative changes in a NetworkSet.

Creates a colored image representing the deviation of each Network from the from mean Network of the NetworkSet, vs frequency.

**Parameters** **m** : int

first s-parameters index

**n** : int

second s-parameter index

**from\_mean** : Boolean

calculate distance from mean if True. or distance from first network in networkset if False.

**operation** : ['\_\_sub\_\_', '\_\_div\_\_', ..]

operation to apply between each network and the reference network, which is either the mean, or the initial ntwk.

**component** : ['s\_mag', 's\_db', 's\_deg' ..]

scalar component of Network to plot on the imshow. should be a property of the Network object.

**vmax** : number

sets upper limit of colorbar, if None, will be set to 3\*mean of the magnitude of the complex difference

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to imshow()

**skrf.networkSet.NetworkSet.uncertainty\_ntwk\_triplet**

`NetworkSet.uncertainty_ntwk_triplet` (*attribute, n\_deviations=3*)

returns a 3-tuple of Network objects which contain the mean, upper\_bound, and lower\_bound for the given Network attribute.

Used to save and plot uncertainty information data

### skrf.networkSet.NetworkSet.write

NetworkSet.**write** (*file=None, \*args, \*\*kwargs*)

Write the NetworkSet to disk using `write()`

**Parameters** **file** : str or file-object

filename or a file-object. If left as None then the filename will be set to Calibration.name, if its not None. If both are None, ValueError is raised.

**\*args, \*\*kwargs** : arguments and keyword arguments

passed through to `write()`

**See Also:**

`skrf.io.general.write`, `skrf.io.general.read`

#### Notes

If the self.name is not None and file is can left as None and the resultant file will have the .ns extension appended to the filename.

#### Examples

```
>>> ns.name = 'my_ns'
>>> ns.write()
```

## 3.4 plotting (skrf.plotting)

This module provides general plotting functions.

### 3.4.1 Plots and Charts

<code>smith([smithR, chart_type, draw_labels, ...])</code>	plots the smith chart of a given radius
<code>plot_smith(z[, smith_r, chart_type, ...])</code>	plot complex data on smith chart
<code>plot_rectangular(x, y[, x_label, y_label, ...])</code>	plots rectangular data and optionally label axes.
<code>plot_polar(theta, r[, x_label, y_label, ...])</code>	plots polar data on a polar plot and optionally label axes.
<code>plot_complex_rectangular(z[, x_label, ...])</code>	plot complex data on the complex plane
<code>plot_complex_polar(z[, x_label, y_label, ...])</code>	plot complex data in polar format.

#### skrf.plotting.smith

skrf.plotting.**smith** (*smithR=1, chart\_type='z', draw\_labels=False, border=False, ax=None*)

plots the smith chart of a given radius

**Parameters** **smithR** : number

radius of smith chart

**chart\_type** : ['z','y']

**Contour type. Possible values are**

- 'z': lines of constant impedance
- 'y': lines of constant admittance

**draw\_labels** : Boolean

annotate real and imaginary parts of impedance on the chart (only if smithR=1)

**border** : Boolean

draw a rectangular border with axis ticks, around the perimeter of the figure. Not used if draw\_labels = True

**ax** : matplotlib.axes object

existing axes to draw smith chart on

### skrf.plotting.plot\_smith

```
skrf.plotting.plot_smith(z, smith_r=1, chart_type='z', x_label='Real', y_label='Imaginary',
                        title='Complex Plane', show_legend=True, axis='equal', ax=None,
                        force_chart=False, *args, **kwargs)
```

plot complex data on smith chart

**Parameters** **z** : array-like, of complex data

data to plot

**smith\_r** : number

radius of smith chart

**chart\_type** : ['z','y']

**Contour type for chart.**

- 'z': lines of constant impedance
- 'y': lines of constant admittance

**x\_label** : string

x-axis label

**y\_label** : string

y-axis label

**title** : string

plot title

**show\_legend** : Boolean

controls the drawing of the legend

**axis\_equal**: Boolean :

sets axis to be equal increments (calls axis('equal'))

**force\_chart** : Boolean

forces the re-drawing of smith chart

**ax** : matplotlib.axes.AxesSubplot object

axes to draw on

**\*args,\*\*kwargs** : passed to `pylab.plot`

**See Also:**

`plot_rectangular` plots rectangular data

`plot_complex_rectangular` plot complex data on complex plane

`plot_polar` plot polar data

`plot_complex_polar` plot complex data on polar plane

`plot_smith` plot complex data on smith chart

### `skrf.plotting.plot_rectangular`

`skrf.plotting.plot_rectangular` (*x*, *y*, *x\_label=None*, *y\_label=None*, *title=None*,  
*show\_legend=True*, *axis='tight'*, *ax=None*, *\*args*, *\*\*kwargs*)  
plots rectangular data and optionally label axes.

**Parameters** *z* : array-like, of complex data

data to plot

**x\_label** : string

x-axis label

**y\_label** : string

y-axis label

**title** : string

plot title

**show\_legend** : Boolean

controls the drawing of the legend

**ax** : `matplotlib.axes.AxesSubplot` object

axes to draw on

**\*args,\*\*kwargs** : passed to `pylab.plot`

### `skrf.plotting.plot_polar`

`skrf.plotting.plot_polar` (*theta*, *r*, *x\_label=None*, *y\_label=None*, *title=None*, *show\_legend=True*,  
*axis\_equal=False*, *ax=None*, *\*args*, *\*\*kwargs*)  
plots polar data on a polar plot and optionally label axes.

**Parameters** *theta* : array-like

data to plot

*r* : array-like

**x\_label** : string

x-axis label

**y\_label** : string



y-axis label

**title** : string

plot title

**show\_legend** : Boolean

controls the drawing of the legend

**ax** : matplotlib.axes.AxesSubplot object

axes to draw on

**\*args, \*\*kwargs** : passed to pylab.plot

**See Also:**

[plot\\_rectangular](#) plots rectangular data

[plot\\_complex\\_rectangular](#) plot complex data on complex plane

[plot\\_polar](#) plot polar data

[plot\\_complex\\_polar](#) plot complex data on polar plane

[plot\\_smith](#) plot complex data on smith chart

### skrf.plotting.plot\_complex\_rectangular

skrf.plotting.**plot\_complex\_rectangular** (*z*, *x\_label*='Real', *y\_label*='Imag', *title*='Complex Plane', *show\_legend*=True, *axis*='equal', *ax*=None, *\*args*, *\*\*kwargs*)

plot complex data on the complex plane

**Parameters** **z** : array-like, of complex data

data to plot

**x\_label** : string

x-axis label

**y\_label** : string

y-axis label

**title** : string

plot title

**show\_legend** : Boolean

controls the drawing of the legend

**ax** : matplotlib.axes.AxesSubplot object

axes to draw on

**\*args, \*\*kwargs** : passed to pylab.plot

**See Also:**

[plot\\_rectangular](#) plots rectangular data

[plot\\_complex\\_rectangular](#) plot complex data on complex plane

`plot_polar` plot polar data  
`plot_complex_polar` plot complex data on polar plane  
`plot_smith` plot complex data on smith chart

### skrf.plotting.plot\_complex\_polar

`skrf.plotting.plot_complex_polar` (*z*, *x\_label=None*, *y\_label=None*, *title=None*, *show\_legend=True*, *axis\_equal=False*, *ax=None*, *\*args*, *\*\*kwargs*)

plot complex data in polar format.

**Parameters**

- z** : array-like, of complex data  
data to plot
- x\_label** : string  
x-axis label
- y\_label** : string  
y-axis label
- title** : string  
plot title
- show\_legend** : Boolean  
controls the drawing of the legend
- ax** : `matplotlib.axes.AxesSubplot` object  
axes to draw on
- \*args, \*\*kwargs** : passed to `pylab.plot`

**See Also:**

`plot_rectangular` plots rectangular data  
`plot_complex_rectangular` plot complex data on complex plane  
`plot_polar` plot polar data  
`plot_complex_polar` plot complex data on polar plane  
`plot_smith` plot complex data on smith chart

## 3.4.2 Misc Functions

<code>save_all_figs</code> ( <i>[dir, format]</i> )	Save all open Figures to disk.
<code>add_markers_to_lines</code> ( <i>[ax, marker_list, ...]</i> )	adds markers to existing lines on a plot
<code>legend_off</code> ( <i>[ax]</i> )	turn off the legend for a given axes.
<code>func_on_all_figs</code> ( <i>func, *args, **kwargs</i> )	runs a function after making all open figures current.

### skrf.plotting.save\_all\_figs

`skrf.plotting.save_all_figs` (*dir*='.', *format*=['eps', 'pdf', 'svg', 'png'])  
 Save all open Figures to disk.

**Parameters** *dir* : string

path to save figures into

**format** : list of strings

the types of formats to save figures as. The elements of this list are passed to **matplotlib**:`'savefig'`. This is a list so that you can save each figure in multiple formats.

### skrf.plotting.add\_markers\_to\_lines

`skrf.plotting.add_markers_to_lines` (*ax*=None, *marker\_list*=['o', 'D', 's', '+', 'x'], *markevery*=10)

adds markers to existing lines on a plot

this is convenient if you have already have a plot made, but then need to add markers afterwards, so that it can be interpreted in black and white. The *markevery* argument makes the markers less frequent than the data, which is generally what you want.

**Parameters** *ax* : matplotlib.Axes

axis which to add markers to, defaults to `gca()`

**marker\_list** : list of marker characters

see `matplotlib.plot` help for possible marker characters

**markevery** : int

markevery number of points with a marker.

### skrf.plotting.legend\_off

`skrf.plotting.legend_off` (*ax*=None)  
 turn off the legend for a given axes.

if no axes is given then it will use current axes.

**Parameters** *ax* : matplotlib.Axes object

axes to operate on

### skrf.plotting.func\_on\_all\_figs

`skrf.plotting.func_on_all_figs` (*func*, *\*args*, *\*\*kwargs*)  
 runs a function after making all open figures current.

useful if you need to change the properties of many open figures at once, like turn off the grid.

**Parameters** *func* : function

function to call

**\*args, \*\*kwargs** : passed to *func*

## Examples

```
>>> rf.func_on_all_figs(grid, alpha=.3)
```

## 3.5 mathFunctions (skrf.mathFunctions)

Provides commonly used mathematical functions.

### 3.5.1 Complex Component Conversion

<code>complex_2_reim(z)</code>	takes:
<code>complex_2_magnitude(input)</code>	returns the magnitude of a complex number.
<code>complex_2_db(input)</code>	returns the magnitude in dB of a complex number.
<code>complex_2_radian(input)</code>	returns the angle complex number in radians.
<code>complex_2_degree(input)</code>	returns the angle complex number in radians.
<code>complex_2_magnitude(input)</code>	returns the magnitude of a complex number.

#### skrf.mathFunctions.complex\_2\_reim

`skrf.mathFunctions.complex_2_reim(z)`

**takes:** input: complex number or array

**return:** real: real part of input imag: imaginary part of input

note: this just calls 'complex\_components'

#### skrf.mathFunctions.complex\_2\_magnitude

`skrf.mathFunctions.complex_2_magnitude(input)`

returns the magnitude of a complex number.

#### skrf.mathFunctions.complex\_2\_db

`skrf.mathFunctions.complex_2_db(input)`

returns the magnitude in dB of a complex number.

**returns:**  $20 \cdot \log_{10}(|z|)$

where z is a complex number

#### skrf.mathFunctions.complex\_2\_radian

`skrf.mathFunctions.complex_2_radian(input)`

returns the angle complex number in radians.

### skrf.mathFunctions.complex\_2\_degree

`skrf.mathFunctions.complex_2_degree(input)`  
returns the angle complex number in radians.

### skrf.mathFunctions.complex\_2\_magnitude

`skrf.mathFunctions.complex_2_magnitude(input)`  
returns the magnitude of a complex number.

## 3.5.2 Phase Unwrapping

<code>unwrap_rad(input)</code>	unwraps a phase given in radians
<code>sqrt_phase_unwrap(input)</code>	takes the square root of a complex number with unwrapped phase

### skrf.mathFunctions.unwrap\_rad

`skrf.mathFunctions.unwrap_rad(input)`  
unwraps a phase given in radians

the normal numpy unwrap is not what you usually want for some reason

### skrf.mathFunctions.sqrt\_phase\_unwrap

`skrf.mathFunctions.sqrt_phase_unwrap(input)`  
takes the square root of a complex number with unwrapped phase

this idea came from Lihan Chen

## 3.5.3 Unit Conversion

<code>radian_2_degree(rad)</code>	
<code>degree_2_radian(deg)</code>	
<code>np_2_db(x)</code>	converts a value in dB to neper's
<code>db_2_np(x)</code>	converts a value in nepers to dB

### skrf.mathFunctions.radian\_2\_degree

`skrf.mathFunctions.radian_2_degree(rad)`

### skrf.mathFunctions.degree\_2\_radian

`skrf.mathFunctions.degree_2_radian(deg)`

### skrf.mathFunctions.np\_2\_db

`skrf.mathFunctions.np_2_db(x)`  
 converts a value in dB to neper's

### skrf.mathFunctions.db\_2\_np

`skrf.mathFunctions.db_2_np(x)`  
 converts a value in nepers to dB

## 3.5.4 Scalar-Complex Conversion

These conversions are useful for wrapping other functions that dont support complex numbers.

<code>complex2Scalar(input)</code>
<code>scalar2Complex(input)</code>

### skrf.mathFunctions.complex2Scalar

`skrf.mathFunctions.complex2Scalar(input)`

### skrf.mathFunctions.scalar2Complex

`skrf.mathFunctions.scalar2Complex(input)`

## 3.5.5 Special Functions

<code>dirac_delta(x)</code>	the dirac function.
<code>neuman(x)</code>	neumans number
<code>null(A[, eps])</code>	calculates the null space of matrix A.

### skrf.mathFunctions.dirac\_delta

`skrf.mathFunctions.dirac_delta(x)`  
 the dirac function.  
 can take numpy arrays or numbers returns 1 or 0

### skrf.mathFunctions.neuman

`skrf.mathFunctions.neuman(x)`  
 neumans number  
 2-dirac\_delta(x)

### skrf.mathFunctions.null

`skrf.mathFunctions.null(A, eps=1e-15)`  
 calculates the null space of matrix A. i found this on stack overflow.

## 3.6 tlineFunctions (skrf.tlineFunctions)

This module provides functions related to transmission line theory.

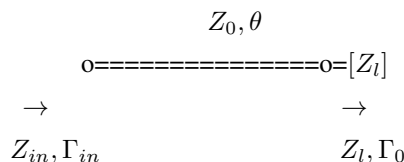
### 3.6.1 Impedance and Reflection Coefficient

These functions relate basic transmission line quantities such as characteristic impedance, input impedance, reflection coefficient, etc. Each function has two names. One is a long-winded but readable name and the other is a short-hand variable-like names. Below is a table relating these two names with each other as well as common mathematical symbols.

Symbol	Variable Name	Long Name
$Z_l$	<code>z_l</code>	load_impedance
$Z_{in}$	<code>z_in</code>	input_impedance
$\Gamma_0$	<code>Gamma_0</code>	reflection_coefficient
$\Gamma_{in}$	<code>Gamma_in</code>	reflection_coefficient_at_theta
$\theta$	<code>theta</code>	electrical_length

There may be a bit of confusion about the difference between the load impedance the input impedance. This is because the load impedance **is** the input impedance at the load. An illustration may provide some useful reference.

Below is a (bad) illustration of a section of uniform transmission line of characteristic impedance  $Z_0$ , and electrical length  $\theta$ . The line is terminated on the right with some load impedance,  $Z_l$ . The input impedance  $Z_{in}$  and input reflection coefficient  $\Gamma_{in}$  are looking in towards the load from the distance  $\theta$  from the load.



So, to clarify the confusion,

$$Z_{in} = Z_l, \quad \Gamma_{in} = \Gamma_l \text{ at } \theta = 0$$

#### Short names

<code>theta(gamma, f, d[, deg])</code>	Calculates the electrical length of a section of transmission line.
<code>z1_2_Gamma0(z0, z1)</code>	Returns the reflection coefficient for a given load impedance, and characteristic impedance.
<code>Gamma0_2_z1(z0, Gamma)</code>	calculates the input impedance given a reflection coefficient and
<code>z1_2_zin(z0, z1, theta)</code>	input impedance of load impedance z1 at a given electrical length,
<code>z1_2_Gamma_in(z0, z1, theta)</code>	
<code>Gamma0_2_Gamma_in(Gamma0, theta)</code>	reflection coefficient at a given electrical length.
<code>Gamma0_2_zin(z0, Gamma0, theta)</code>	calculates the input impedance at electrical length theta, given a

**skrf.tlineFunctions.theta**`skrf.tlineFunctions.theta` (*gamma*, *f*, *d*, *deg=False*)

Calculates the electrical length of a section of transmission line.

$$\theta = \gamma(f) \cdot d$$

**Parameters** **gamma** : function

propagation constant function, which takes frequency in hz as a sole argument. see Notes.

**l** : number or array-like

length of line, in meters

**f** : number or array-like

frequency at which to calculate

**deg** : Boolean

return in degrees or not.

**Returns** **theta** : number or array-like

electrical length in radians or degrees, depending on value of deg.

**See Also:**[electrical\\_length\\_2\\_distance](#) opposite conversion**Notes**

the convention has been chosen that forward propagation is represented by the positive imaginary part of the value returned by the gamma function

**skrf.tlineFunctions.zl\_2\_Gamma0**`skrf.tlineFunctions.zl_2_Gamma0` (*z0*, *zl*)

Returns the reflection coefficient for a given load impedance, and characteristic impedance.

For a transmission line of characteristic impedance  $Z_0$  terminated with load impedance  $Z_l$ , the complex reflection coefficient is given by,

$$\Gamma = \frac{Z_l - Z_0}{Z_l + Z_0}$$

**Parameters** **z0** : number or array-like

characteristic impedance

**zl** : number or array-like

load impedance (aka input impedance)

**Returns** **gamma** : number or array-like



reflection coefficient

**See Also:**

`Gamma0_2_z1` reflection coefficient to load impedance

**Notes**

inputs are typecasted to 1D complex array

**skrf.tlineFunctions.Gamma0\_2\_z1**

`skrf.tlineFunctions.Gamma0_2_z1` ( $z0$ ,  $\Gamma$ )  
calculates the input impedance given a reflection coefficient and characteristic impedance

$$Z_0 \left( \frac{1 + \Gamma}{1 - \Gamma} \right)$$

**Parameters**  $\Gamma$  : number or array-like

complex reflection coefficient

$z0$  : number or array-like

characteristic impedance

**Returns**  $zin$  : number or array-like

input impedance

**skrf.tlineFunctions.zl\_2\_zin**

`skrf.tlineFunctions.zl_2_zin` ( $z0$ ,  $zl$ ,  $\theta$ )  
input impedance of load impedance  $zl$  at a given electrical length, given characteristic impedance  $z0$ .

**Parameters**  $z0$  : characteristic impedance.

$zl$  : load impedance

$\theta$  : electrical length of the line, (may be complex)

**skrf.tlineFunctions.zl\_2\_Gamma\_in**

`skrf.tlineFunctions.zl_2_Gamma_in` ( $z0$ ,  $zl$ ,  $\theta$ )

**skrf.tlineFunctions.Gamma0\_2\_Gamma\_in**

`skrf.tlineFunctions.Gamma0_2_Gamma_in` ( $\Gamma_0$ ,  $\theta$ )  
reflection coefficient at a given electrical length.

$$\Gamma_{in} = \Gamma_0 e^{-2j\theta}$$

**Parameters**  $\Gamma_0$  : number or array-like

reflection coefficient at theta=0

**theta** : number or array-like

electrical length, (may be complex)

**Returns** **Gamma\_in** : number or array-like

input reflection coefficient

### skrf.tlineFunctions.Gamma0\_2\_zin

skrf.tlineFunctions.**Gamma0\_2\_zin** (*z0*, *Gamma0*, *theta*)

calculates the input impedance at electrical length theta, given a reflection coefficient and characteristic impedance of the medium Parameters ———

*z0* - characteristic impedance. *Gamma*: reflection coefficient *theta*: electrical length of the line, (may be complex)

**returns** *zin*: input impedance at theta

### Long-names

<code>distance_2_electrical_length(gamma, f, d[, deg])</code>	Calculates the electrical length of a section of transmission line.
<code>electrical_length_2_distance(theta, gamma, f0)</code>	Convert electrical length to a physical distance.
<code>reflection_coefficient_at_theta(Gamma0, theta)</code>	reflection coefficient at a given electrical length.
<code>reflection_coefficient_2_input_impedance(z0, ...)</code>	calculates the input impedance given a reflection coefficient and characteristic impedance.
<code>reflection_coefficient_2_input_impedance_at_theta(z0, ...)</code>	calculates the input impedance at electrical length theta.
<code>input_impedance_at_theta(z0, z1, theta)</code>	input impedance of load impedance z1 at a given electrical length theta.
<code>load_impedance_2_reflection_coefficient(z0, z1)</code>	Returns the reflection coefficient for a given load impedance.
<code>load_impedance_2_reflection_coefficient_at_theta(z0, ...)</code>	Returns the reflection coefficient for a given load impedance at a given electrical length.

### skrf.tlineFunctions.distance\_2\_electrical\_length

skrf.tlineFunctions.**distance\_2\_electrical\_length** (*gamma*, *f*, *d*, *deg=False*)

Calculates the electrical length of a section of transmission line.

$$\theta = \gamma(f) \cdot d$$

**Parameters** **gamma** : function

propagation constant function, which takes frequency in hz as a sole argument. see Notes.

**l** : number or array-like

length of line, in meters

**f** : number or array-like

frequency at which to calculate

**deg** : Boolean

return in degrees or not.

**Returns** **theta** : number or array-like

electrical length in radians or degrees, depending on value of deg.

**See Also:**

[electrical\\_length\\_2\\_distance](#) opposite conversion

**Notes**

the convention has been chosen that forward propagation is represented by the positive imaginary part of the value returned by the gamma function

### `skrf.tlineFunctions.electrical_length_2_distance`

`skrf.tlineFunctions.electrical_length_2_distance(theta, gamma, f0, deg=True)`

Convert electrical length to a physical distance.

$$d = \frac{\theta}{\gamma(f_0)}$$

**Parameters** **theta** : number or array-like

electical length. units depend on *deg* option

**gamma** : function

propagation constant function, which takes frequency in hz as a sole argument. see Notes

**f0** : number or array-like

frequency at which to calculate

**deg** : Boolean

return in degrees or not.

**Returns** **d**: physical distance :

**See Also:**

[distance\\_2\\_electrical\\_length](#) opposite conversion

**Notes**

the convention has been chosen that forward propagation is represented by the positive imaginary part of the value returned by the gamma function

### `skrf.tlineFunctions.reflection_coefficient_at_theta`

`skrf.tlineFunctions.reflection_coefficient_at_theta(Gamma0, theta)`

reflection coefficient at a given electrical length.

$$\Gamma_{in} = \Gamma_0 e^{-2j\theta}$$

**Parameters** **Gamma0** : number or array-like  
 reflection coefficient at theta=0

**theta** : number or array-like  
 electrical length, (may be complex)

**Returns** **Gamma\_in** : number or array-like  
 input reflection coefficient

**skrf.tlineFunctions.reflection\_coefficient\_2\_input\_impedance**

`skrf.tlineFunctions.reflection_coefficient_2_input_impedance` (*z0*, *Gamma*)  
 calculates the input impedance given a reflection coefficient and characterisitc impedance

$$Z_0 \left( \frac{1 + \Gamma}{1 - \Gamma} \right)$$

**Parameters** **Gamma** : number or array-like  
 complex reflection coefficient

**z0** : number or array-like  
 characteristic impedance

**Returns** **zin** : number or array-like  
 input impedance

**skrf.tlineFunctions.reflection\_coefficient\_2\_input\_impedance\_at\_theta**

`skrf.tlineFunctions.reflection_coefficient_2_input_impedance_at_theta` (*z0*,  
*Gamma0*,  
*theta*)  
 calculates the input impedance at electrical length theta, given a reflection coefficient and characterisitc impedance of the medium Parameters ———

*z0* - characteristic impedance. *Gamma*: reflection coefficient *theta*: electrical length of the line, (may be complex)

**returns** *zin*: input impedance at theta

**skrf.tlineFunctions.input\_impedance\_at\_theta**

`skrf.tlineFunctions.input_impedance_at_theta` (*z0*, *zl*, *theta*)  
 input impedance of load impedance *zl* at a given electrical length, given characteristic impedance *z0*.

**Parameters** **z0** : characteristic impedance.

**zl** : load impedance

**theta** : electrical length of the line, (may be complex)

**skrf.tlineFunctions.load\_impedance\_2\_reflection\_coefficient**

`skrf.tlineFunctions.load_impedance_2_reflection_coefficient` (*z0*, *zl*)

Returns the reflection coefficient for a given load impedance, and characteristic impedance.

For a transmission line of characteristic impedance  $Z_0$  terminated with load impedance  $Z_l$ , the complex reflection coefficient is given by,

$$\Gamma = \frac{Z_l - Z_0}{Z_l + Z_0}$$

**Parameters** **z0** : number or array-like

characteristic impedance

**zl** : number or array-like

load impedance (aka input impedance)

**Returns** **gamma** : number or array-like

reflection coefficient

**See Also:**

[Gamma0\\_2\\_z1](#) reflection coefficient to load impedance

**Notes**

inputs are typecasted to 1D complex array

**skrf.tlineFunctions.load\_impedance\_2\_reflection\_coefficient\_at\_theta**

`skrf.tlineFunctions.load_impedance_2_reflection_coefficient_at_theta` (*z0*, *zl*,  
*theta*)

**3.6.2 Distributed Circuit and Wave Quantities**


---

<code>distributed_circuit_2_propagation_impedance(...)</code>	Converts distributed circuit values to wave quantities.
<code>propagation_impedance_2_distributed_circuit(...)</code>	Converts wave quantities to distributed circuit values.

---

**skrf.tlineFunctions.distributed\_circuit\_2\_propagation\_impedance**

`skrf.tlineFunctions.distributed_circuit_2_propagation_impedance` (*distributed\_admittance*,  
*distributed\_impedance*)

Converts distributed circuit values to wave quantities.

This converts complex distributed impedance and admittance to propagation constant and characteristic

impedance. The relation is

$$Z_0 = \sqrt{\frac{Z'}{Y'}} \quad \gamma = \sqrt{Z'Y'}$$

**Parameters** `distributed_admittance` : number, array-like

distributed admittance

`distributed_impedance` : number, array-like

distributed impedance

**Returns** `propagation_constant` : number, array-like

distributed impedance

`characteristic_impedance` : number, array-like

distributed impedance

**See Also:**

[propagation\\_impedance\\_2\\_distributed\\_circuit](#) opposite conversion

### `skrf.tlineFunctions.propagation_impedance_2_distributed_circuit`

`skrf.tlineFunctions.propagation_impedance_2_distributed_circuit` (*propagation\_constant*, *characteristic\_impedance*)

Converts wave quantities to distributed circuit values.

Converts complex propagation constant and characteristic impedance to distributed impedance and admittance. The relation is,

$$Z' = \gamma Z_0 \quad Y' = \frac{\gamma}{Z_0}$$

**Parameters** `propagation_constant` : number, array-like

distributed impedance

`characteristic_impedance` : number, array-like

distributed impedance

**Returns** `distributed_admittance` : number, array-like

distributed admittance

`distributed_impedance` : number, array-like

distributed impedance

**See Also:**

[distributed\\_circuit\\_2\\_propagation\\_impedance](#) opposite conversion

## 3.6.3 Transmission Line Physics

<code>skin_depth(f, rho, mu_r)</code>	the skin depth for a material.
<code>surface_resistivity(f, rho, mu_r)</code>	surface resistivity.

### `skrf.tlineFunctions.skin_depth`

`skrf.tlineFunctions.skin_depth` (*f*, *rho*, *mu\_r*)  
the skin depth for a material.

see [www.microwaves101.com](http://www.microwaves101.com) for more info.

**Parameters** **f**: number or array-like

frequency, in Hz

**rho**: number or array-like

bulk resistivity of material, in ohm\*m

**mu\_r**: number or array-like

relative permeability of material

**Returns** **skin depth**: number or array-like

the skin depth, in m

### `skrf.tlineFunctions.surface_resistivity`

`skrf.tlineFunctions.surface_resistivity` (*f*, *rho*, *mu\_r*)  
surface resistivity.

see [www.microwaves101.com](http://www.microwaves101.com) for more info.

**Parameters** **f**: number or array-like

frequency, in Hz

**rho**: number or array-like

bulk resistivity of material, in ohm\*m

**mu\_r**: number or array-like

relative permeability of material

**Returns** **surface resistivity: ohms/square**:

## 3.7 constants (`skrf.constants`)

This module contains pre-initialized objects's.

### 3.7.1 Standard Waveguide Bands

#### Frequency Objects

These are predefined `Frequency` objects that correspond to standard waveguide bands. This information is taken from the VDI Application Note 1002<sup>31</sup>.

Object Name	Description
<code>f_wr10</code>	WR-10, 75-110 GHz
<code>f_wr3</code>	WR-3, 220-325 GHz
<code>f_wr2p2</code>	WR-2.2, 330-500 GHz
<code>f_wr1p5</code>	WR-1.5, 500-750 GHz
<code>f_wr1</code>	WR-1, 750-1100 GHz
...	...

#### RectangularWaveguide Objects

These are predefined `RectangularWaveguide` objects for standard waveguide bands.

Object Name	Description
<code>wr10</code>	WR-10, 75-110 GHz
<code>wr3</code>	WR-3, 220-325 GHz
<code>wr2p2</code>	WR-2.2, 330-500 GHz
<code>wr1p5</code>	WR-1.5, 500-750 GHz
<code>wr1</code>	WR-1, 750-1100 GHz
...	...

### 3.7.2 Shorthand Names

Below is a list of shorthand object names which can be used to save some typing. These names are defined in the main `__init__` module.

Shorthand	Full Object Name
<code>F</code>	<code>Frequency</code>
<code>N</code>	<code>Network</code>
<code>NS</code>	<code>NetworkSet</code>
<code>M</code>	<code>Media</code>
<code>C</code>	<code>Calibration</code>

The following are shorthand names for commonly used, but unfortunately longwinded functions.

Shorthand	Full Object Name
<code>saf</code>	<code>save_all_figs()</code>

### 3.7.3 References

## 3.8 util (`skrf.util`)

Holds utility functions that are general conveniences.

---

<sup>31</sup> VDI Application Note: VDI Waveguide Band Designations (VDI-1002) <http://vadiodes.com/VDI/pdf/waveguidechart200908.pdf>



### 3.8.1 General

<code>now_string()</code>	returns a unique sortable string, representing the current time
<code>find_nearest(array, value)</code>	find nearest value in array.
<code>find_nearest_index(array, value)</code>	find nearest value in array.
<code>get_fid(file, *args, **kwargs)</code>	Returns a file object, given a filename or file object
<code>get_extn(filename)</code>	Get the extension from a filename.

#### `skrf.util.now_string`

`skrf.util.now_string()`  
 returns a unique sortable string, representing the current time  
 nice for generating date-time stamps to be used in file-names

#### `skrf.util.find_nearest`

`skrf.util.find_nearest(array, value)`  
 find nearest value in array.  
 taken from <http://stackoverflow.com/questions/2566412/find-nearest-value-in-numpy-array>

**Parameters** `array` : `numpy.ndarray`  
 array we are searching for a value in  
**value** : element of the array  
 value to search for  
**Returns** `found_value` : an element of the array  
 the value that is numerically closest to *value*

#### `skrf.util.find_nearest_index`

`skrf.util.find_nearest_index(array, value)`  
 find nearest value in array.  
**Parameters** `array` : `numpy.ndarray`  
 array we are searching for a value in  
**value** : element of the array  
 value to search for  
**Returns** `found_index` : `int`  
 the index at which the numerically closest element to *value* was found at  
**taken from** <http://stackoverflow.com/questions/2566412/find-nearest-value-in-numpy-array>  
**array** :

### skrf.util.get\_fid

`skrf.util.get_fid(file, *args, **kwargs)`

Returns a file object, given a filename or file object

Useful when you want to allow the arguments of a function to be either files or filenames

**Parameters** `file` : str or file-object

file to open

**\*args, \*\*kwargs** : arguments and keyword arguments

passed through to `pickle.load`

### skrf.util.get\_extn

`skrf.util.get_extn(filename)`

Get the extension from a filename.

The extension is defined as everything passed the last `'.'`. Returns `None` if it aint got one

**Parameters** `filename` : string

the filename

**Returns** `ext` : string, `None`

either the extension (not including `'.'`) or `None` if there isnt one

## 3.9 io (skrf.io)

This Package provides functions and objects for input/output.

The general functions `read()` and `write()` can be used to read and write [almost] any skrf object to disk, using the `pickle` module.

Reading and writing touchstone files is supported through the `Touchstone` class, which can be more easily used through the Network constructor, `__init__()`

### 3.9.1 general (skrf.io.general)

General io functions for reading and writing skrf objects

<code>read(file, *args, **kwargs)</code>	Read skrf object[s] from a pickle file
<code>read_all([dir, contains])</code>	Read all skrf objects in a directory
<code>write(file, obj[, overwrite])</code>	Write skrf object[s] to a file
<code>write_all(dict_objs[, dir])</code>	Write a dictionary of skrf objects individual files in <i>dir</i> .
<code>save_sesh(dict_objs[, file, module, ...])</code>	Save all <i>skrf</i> objects in the local namespace.

#### skrf.io.general.read

`skrf.io.general.read(file, *args, **kwargs)`

Read skrf object[s] from a pickle file

Reads a skrf object that is written with `write()`, which uses the `pickle` module.

**Parameters** `file` : str or file-object  
 name of file, or a file-object  
`*args, **kwargs` : arguments and keyword arguments  
 passed through to `pickle.load`

**See Also:**

`read` read a skrf object  
`write` write skrf object[s]  
`read_all` read all skrf objects in a directory  
`write_all` write dictionary of skrf objects to a directory

**Notes**

if `file` is a file-object it is left open, if it is a filename then a file-object is opened and closed. If file is a file-object and reading fails, then the position is reset back to 0 using `seek` if possible.

**Examples**

```
>>> n = rf.Network(f=[1,2,3],s=[1,1,1],z0=50)
>>> n.write('my_ntwk.ntwk')
>>> n_2 = rf.read('my_ntwk.ntwk')
```

**skrf.io.general.read\_all**

`skrf.io.general.read_all` (`dir='.'`, `contains=None`)

Read all skrf objects in a directory

Attempts to load all files in `dir`, using `read()`. Any file that is not readable by skrf is skipped. Optionally, simple filtering can be achieved through the use of `contains` argument.

**Parameters** `dir` : str, optional  
 the directory to load from, default `'.'`  
`contains` : str, optional  
 if not `None`, only files containing this substring will be loaded

**Returns** `out` : dictionary  
 dictionary containing all loaded skrf objects. keys are the filenames without extensions, and the values are the objects

**See Also:**

`read` read a skrf object  
`write` write skrf object[s]  
`read_all` read all skrf objects in a directory  
`write_all` write dictionary of skrf objects to a directory

## Examples

```
>>> rf.read_all('skrf/data/')
{'delay_short': 1-Port Network: 'delay_short', 75-110 GHz, 201 pts, z0=[ 50.+0.j],
'line': 2-Port Network: 'line', 75-110 GHz, 201 pts, z0=[ 50.+0.j 50.+0.j],
'ntwk1': 2-Port Network: 'ntwk1', 1-10 GHz, 91 pts, z0=[ 50.+0.j 50.+0.j],
'one_port': one port Calibration: 'one_port', 500-750 GHz, 201 pts, 4-ideals/4-measured,
...

```

## skrf.io.general.write

skrf.io.general.**write** (*file*, *obj*, *overwrite=True*)

Write skrf object[s] to a file

This uses the `pickle` module to write skrf objects to a file. Note that you can write any pickl-able python object. For example, you can write a list or dictionary of `Network` objects or `Calibration` objects. This will write out a single file. If you would like to write out a seperate file for each object, use `write_all()`.

**Parameters** **file** : file or string

File or filename to which the data is saved. If file is a file-object, then the filename is unchanged. If file is a string, an appropriate extension will be appended to the file name if it does not already have an extension.

**obj** : an object, or list/dict of objects

object or list/dict of objects to write to disk

**overwrite** : Boolean

if file exists, should it be overwritten?

**See Also:**

**read** read a skrf object

**write** write skrf object[s]

**read\_all** read all skrf objects in a directory

**write\_all** write dictionary of skrf objects to a directory

**skrf.network.Network.write** write method of Network

**skrf.calibration.calibration.Calibration.write** write method of Calibration

## Notes

If *file* is a str, but doesnt contain a suffix, one is chosen automatically. Here are the extensions

skrf object	extension
Frequency	'.freq'
Network	'.ntwk'
NetworkSet	'.ns'
Calibration	'.cal'
Media	'.med'
other	'.p'

To make file written by this method cross-platform, the pickling protocol 2 is used. See `pickle` for more info.

## Examples

Convert a touchstone file to a pickled Network,

```
>>> n = rf.Network('my_ntwk.s2p')
>>> rf.write('my_ntwk', n)
>>> n_red = rf.read('my_ntwk.ntwk')
```

Writing a list of different objects

```
>>> n = rf.Network('my_ntwk.s2p')
>>> ns = rf.NetworkSet([n,n,n])
>>> rf.write('out', [n,ns])
>>> n_red = rf.read('out.p')
```

## skrf.io.general.write\_all

`skrf.io.general.write_all` (*dict\_objs*, *dir*='.', \**args*, \*\**kwargs*)

Write a dictionary of skrf objects individual files in *dir*.

Each object is written to its own file. The filename used for each object is taken from its key in the dictionary. If no extension exists in the key, then one is added. See `write()` for a list of extensions. If you would like to write the dictionary to a single output file use `write()`.

**Parameters** `dict_objs` : dict

dictionary of skrf objects

`dir` : str

directory to save skrf objects into

`*args, **kwargs` : :

passed through to `write()`. *overwrite* option may be of use.

**See Also:**

`read` read a skrf object

`write` write skrf object[s]

`read_all` read all skrf objects in a directory

`write_all` write dictionary of skrf objects to a directory

## Notes

Any object in `dict_objs` that is pickl-able will be written.

## Examples

Writing a diction of different skrf objects

```
>>> from skrf.data import line, short
>>> d = {'ring_slot':ring_slot, 'one_port_cal':one_port_cal}
>>> rf.write_all(d)
```

### skrf.io.general.save\_sesh

`skrf.io.general.save_sesh(dict_objs, file='skrfSesh.p', module='skrf', exclude_prefix='_')`  
Save all *skrf* objects in the local namespace.

This is used to save current workspace in a hurry, by passing it the output of `locals()` (see Examples). Note this can be used for other modules as well by passing a different *module* name.

**Parameters** `dict_objs` : dict

dictionary containing *skrf* objects. See the Example.

**file** : str or file-object, optional

the file to save all objects to

**module** : str, optional

the module name to grep for.

**exclude\_prefix**: str, optional :

don't save objects which have this as a prefix.

**See Also:**

**read** read a skrf object

**write** write skrf object[s]

**read\_all** read all skrf objects in a directory

**write\_all** write dictionary of skrf objects to a directory

### Examples

Write out all skrf objects in current namespace.

```
>>> rf.write_all(locals(), 'mysesh.p')
```

## 3.9.2 touchstone (skrf.io.touchstone)

Touchstone class

---

`Touchstone(file)` class to read touchstone s-parameter files

---

### skrf.io.touchstone.Touchstone

**class** `skrf.io.touchstone.Touchstone(file)`  
class to read touchstone s-parameter files

The reference for writing this class is the draft of the Touchstone(R) File Format Specification Rev 2.0<sup>32</sup>

### Methods

---

<sup>32</sup> [http://www.eda-stds.org/ibis/adhoc/interconnect/touchstone\\_spec2\\_draft.pdf](http://www.eda-stds.org/ibis/adhoc/interconnect/touchstone_spec2_draft.pdf)

<code>__init__</code>	constructor
<code>get_comments</code>	Returns the comments which appear anywhere in the file.
<code>get_format</code>	returns the file format string used for the given format.
<code>get_noise_data</code>	TODO: NIY
<code>get_noise_names</code>	TODO: NIY
<code>get_sparameter_arrays</code>	returns the sparameters as a tuple of arrays, where the first element is
<code>get_sparameter_data</code>	get the data of the sparameter with the given format.
<code>get_sparameter_names</code>	generate a list of column names for the s-parameter data
<code>load_file</code>	Load the touchstone file into the internal data structures

### `skrf.io.touchstone.Touchstone.__init__`

`Touchstone.__init__(file)`

constructor

**Parameters** `file` : str or file-object

touchstone file to load

#### Examples

From filename

```
>>> t = rf.Touchstone('network.s2p')
```

From file-object

```
>>> file = open('network.s2p')
>>> t = rf.Touchstone(file)
```

### `skrf.io.touchstone.Touchstone.get_comments`

`Touchstone.get_comments(ignored_comments=['Created with skrf'])`

Returns the comments which appear anywhere in the file. Comment lines containing ignored comments are removed. By default these are comments which contain special meaning withing skrf and are not user comments.

### `skrf.io.touchstone.Touchstone.get_format`

`Touchstone.get_format(format='ri')`

returns the file format string used for the given format. This is usefull to get some informations.

### `skrf.io.touchstone.Touchstone.get_noise_data`

`Touchstone.get_noise_data()`

TODO: NIY

### `skrf.io.touchstone.Touchstone.get_noise_names`

`Touchstone.get_noise_names()`

TODO: NIY

### skrf.io.touchstone.Touchstone.get\_sparameter\_arrays

Touchstone.**get\_sparameter\_arrays** ()

returns the sparameters as a tuple of arrays, where the first element is the frequency vector (in Hz) and the s-parameters are a 3d numpy array. The values of the sparameters are complex number. usage:

```
f,a = self.sgetparameter_arrays() s11 = a[:,0,0]
```

### skrf.io.touchstone.Touchstone.get\_sparameter\_data

Touchstone.**get\_sparameter\_data** (*format='ri'*)

get the data of the sparameter with the given format. supported formats are:

orig: unmodified s-parameter data ri: data in real/imaginary ma: data in magnitude and angle (degree)  
db: data in log magnitude and angle (degree)

Returns a list of numpy.arrays

### skrf.io.touchstone.Touchstone.get\_sparameter\_names

Touchstone.**get\_sparameter\_names** (*format='ri'*)

generate a list of column names for the s-parameter data The names are different for each format. possible format parameters:

ri, ma, db, orig (where orig refers to one of the three others)

returns a list of strings.

### skrf.io.touchstone.Touchstone.load\_file

Touchstone.**load\_file** (*fid*)

Load the touchstone file into the internal data structures

Functions related to reading/writing touchstones.

<code>hfss_touchstone_2_gamma_z0(filename)</code>	Extracts Z0 and Gamma comments from touchstone file
<code>hfss_touchstone_2_media(filename[, f_unit])</code>	Creates a <code>Media</code> object from a HFSS-style touchstone file with Gamma and Z0 comments

### skrf.io.touchstone.hfss\_touchstone\_2\_gamma\_z0

skrf.io.touchstone.**hfss\_touchstone\_2\_gamma\_z0** (*filename*)

Extracts Z0 and Gamma comments from touchstone file

Takes a HFSS-style touchstone file with Gamma and Z0 comments and extracts a triplet of arrays being: (frequency, Gamma, Z0)

**Parameters** `filename` : string

the HFSS-style touchstone file

**Returns** `f` : numpy.ndarray

frequency vector (in Hz)

`gamma` : complex numpy.ndarray



complex propagation constant  
**z0** : numpy.ndarray  
 complex port impedance

### Examples

```
>>> f, gamm, z0 = rf.hfss_touchstone_2_gamma_z0('line.s2p')
```

### skrf.io.touchstone.hfss\_touchstone\_2\_media

`skrf.io.touchstone.hfss_touchstone_2_media(filename, f_unit='ghz')`  
 Creates a `Media` object from a HFSS-style touchstone file with Gamma and Z0 comments

**Parameters** `filename` : string

the HFSS-style touchstone file

**f\_unit** : ['hz', 'khz', 'mhz', 'ghz']

passed to `f_unit` parameters of Frequency constructor

**Returns** `my_media` : `skrf.media.Media` object

the transmission line model defined by the gamma, and z0 comments in the HFSS file.

### See Also:

`hfss_touchstone_2_gamma_z0` returns gamma, and z0

### Examples

```
>>> port1_media, port2_media = rf.hfss_touchstone_2_media('line.s2p')
```

## 3.9.3 csv (skrf.io.csv)

Functions for reading and writing standard csv files

<code>read_pna_csv(filename, *args, **kwargs)</code>	Reads data from a csv file written by an Agilent PNA
<code>pna_csv_2_ntwks(filename)</code>	Reads a PNAX csv file, and returns a list of one-port Networks

### skrf.io.csv.read\_pna\_csv

`skrf.io.csv.read_pna_csv(filename, *args, **kwargs)`  
 Reads data from a csv file written by an Agilent PNA

**Parameters** `filename` : str

the file

**\*args, \*\*kwargs** :

**Returns** `header` : str

The header string, which is the line following the 'BEGIN'

**comments** : str

All lines that begin with a ‘!’

**data** : `numpy.ndarray`

An array containing the data. The meaning of which depends on the header.

**See Also:**

`pna_csv_2_ntwks` Reads a csv file which contains s-parameter data

**Examples**

```
>>> header, comments, data = rf.read_pna_csv('myfile.csv')
```

### `skrf.io.csv.pna_csv_2_ntwks`

`skrf.io.csv.pna_csv_2_ntwks` (*filename*)

Reads a PNAX csv file, and returns a list of one-port Networks

Note this only works if csv is save in Real/Imaginary format for now

**Parameters** **filename** : str

filename

**Returns** **out** : list of `Network` objects

list of Networks representing the data contained in column pairs

## 3.10 calibration (`skrf.calibration`)

This Package provides a high-level class representing a calibration instance, as well as calibration algorithms and supporting functions.

Both one and two port calibrations are supported. These calibration algorithms allow for redundant measurements, by using a simple least squares estimator to solve for the embedding network.

### 3.10.1 Modules

`calibration` (`skrf.calibration.calibration`)

Contains the Calibration class, and supporting functions

**Calibration Class**

---

`Calibration`(*measured*, *ideals*[, *type*, ...]) An object to represent a VNA calibration instance.

---

`skrf.calibration.calibration.Calibration`

```
class skrf.calibration.calibration.Calibration (measured, ideals, type=None,
                                                is_reciprocal=False, name=None,
                                                sloppy_input=False, **kwargs)
```

An object to represent a VNA calibration instance.

A Calibration object is used to perform a calibration given a set measurements and ideals responses. It can run a calibration, store results, and apply the results to calculate corrected measurements.

### Attributes

<code>Ts</code>	T-matrices used for de-embedding, a two-port calibration.
<code>caled_ntwk_sets</code>	returns a NetworkSet for each caled_ntwk, based on their names
<code>caled_ntwks</code>	list of the calibrated, calibration standards.
<code>calibration_algorithm_dict</code>	
<code>coefs</code>	coefs: a dictionary holding the calibration coefficients
<code>error_ntwk</code>	A Network object which represents the error network being
<code>nports</code>	the number of ports in the calibration
<code>nstandards</code>	number of ideal/measurement pairs in calibration
<code>output_from_cal</code>	a dictionary holding all of the output from the calibration
<code>residual_ntwks</code>	returns a the residuals for each calibration standard in the
<code>residuals</code>	if calibration is overdetermined, this holds the residuals
<code>type</code>	string representing what type of calibration is to be

#### skrf.calibration.calibration.Calibration.Ts

Calibration.**Ts**

T-matrices used for de-embedding, a two-port calibration.

#### skrf.calibration.calibration.Calibration.caled\_ntwk\_sets

Calibration.**caled\_ntwk\_sets**

returns a NetworkSet for each caled\_ntwk, based on their names

#### skrf.calibration.calibration.Calibration.caled\_ntwks

Calibration.**caled\_ntwks**

list of the calibrated, calibration standards.

#### skrf.calibration.calibration.Calibration.calibration\_algorithm\_dict

Calibration.**calibration\_algorithm\_dict** = {'two port': <function two\_port at 0x4b598c0>, 'one port parametric': dictionary holding calibration algorithms.

#### skrf.calibration.calibration.Calibration.coefs

Calibration.**coefs**

coefs: a dictionary holding the calibration coefficients

for one port cal's 'directivity':e00 'reflection tracking':e01e10 'source match':e11

for 7-error term two port cal's TODO:

#### skrf.calibration.calibration.Calibration.error\_ntwk

Calibration.**error\_ntwk**

A Network object which represents the error network being calibrated out.

**skrf.calibration.calibration.Calibration.nports**

`Calibration.nports`

the number of ports in the calibration

**skrf.calibration.calibration.Calibration.nstandards**

`Calibration.nstandards`

number of ideal/measurement pairs in calibration

**skrf.calibration.calibration.Calibration.output\_from\_cal**

`Calibration.output_from_cal`

a dictionary holding all of the output from the calibration algorithm

**skrf.calibration.calibration.Calibration.residual\_ntwks**

`Calibration.residual_ntwks`

returns a the residuals for each calibration standard in the form of a list of Network types.

these residuals are calculated in the ‘calibrated domain’, meaning they are

$$r = (E.inv ** m - i)$$

**where,** r: residual network, E: embedding network, m: measured network i: ideal network

This way the units of the residual networks are meaningful

**note:** the residuals are only calculated if they are not existent.

so, if you want to re-calculate the residual networks then you delete the property ‘\_residual\_ntwks’.

**skrf.calibration.calibration.Calibration.residuals**

`Calibration.residuals`

if calibration is overdetermined, this holds the residuals in the form of a vector.

also available are the complex residuals in the form of skrf.Network’s, see the property ‘residual\_ntwks’

**from numpy.linalg.lstsq:** residues: the sum of the residues; squared euclidean norm for each column vector in b (given ax=b)

**skrf.calibration.calibration.Calibration.type**

`Calibration.type`

string representing what type of calibration is to be performed. supported types at the moment are:

‘one port’: standard one-port cal. if more than 2 measurement/ideal pairs are given it will calculate the least squares solution.

‘two port’: two port calibration based on the error-box model

note: algorithms referenced by calibration\_algorithm\_dict, are stored in calibrationAlgorithms.py

**Methods**

<code>__init__</code>	Calibration initializer.
<code>apply_cal</code>	apply the current calibration to a measurement.
Continued on next page	

Table 3.35 – continued from previous page

<code>apply_cal_to_all_in_dir</code>	convenience function to apply calibration to an entire directory
<code>biased_error</code>	estimate of biased error for overdetermined calibration with
<code>func_per_standard</code>	
<code>mean_residuals</code>	
<code>plot_coefs_db</code>	plot magnitude of the error coefficient dictionary
<code>plot_errors</code>	plot calibration error metrics for an over-determined calibration.
<code>plot_residuals</code>	plots a component of the residual errors on the Calibration-plane.
<code>plot_residuals_db</code>	see <code>plot_residuals</code>
<code>plot_residuals_mag</code>	see <code>plot_residuals</code>
<code>plot_residuals_smith</code>	see <code>plot_residuals</code>
<code>plot_uncertainty_per_standard</code>	Plots uncertainty associated with each calibration standard.
<code>run</code>	runs the calibration algorithm.
<code>total_error</code>	estimate of total error for overdetermined calibration with
<code>unbiased_error</code>	estimate of unbiased error for overdetermined calibration with
<code>uncertainty_per_standard</code>	given that you have repeat-connections of single standard,
<code>write</code>	Write the Calibration to disk using <code>write()</code>

**skrf.calibration.calibration.Calibration.\_\_init\_\_**

`Calibration.__init__(measured, ideals, type=None, is_reciprocal=False, name=None, sloppy_input=False, **kwargs)`

Calibration initializer.

**Parameters** `measured` : list of `Network` objects

Raw measurements of the calibration standards. The order must align with the `ideals` parameter

`ideals` : list of `Network` objects

Predicted ideal response of the calibration standards. The order must align with `ideals` list

**Notes**

All calibration algorithms are in stored in `skrf.calibration.calibrationAlgorithms`, refer to that file for documentation on the algorithms themselves. The Calibration class accesses those functions through the attribute `'calibration_algorithm_dict'`.

**References****Examples**

See the [Calibration](#) tutorial, or the examples sections for [One-Port Calibration](#) and [../../../../examples/twoport\\_calibration](#)

**skrf.calibration.calibration.Calibration.apply\_cal**

`Calibration.apply_cal(input_ntwk)`

apply the current calibration to a measurement.

**takes:**

`input_ntwk`: the measurement to apply the calibration to, a `Network` type.

**returns:** `caled`: the calibrated measurement, a Network type.

**skrf.calibration.calibration.Calibration.apply\_cal\_to\_all\_in\_dir**

`Calibration.apply_cal_to_all_in_dir` (*dir*='.', *contains*=None, *f\_unit*='ghz')

convenience function to apply calibration to an entire directory of measurements, and return a dictionary of the calibrated results, optionally the user can 'grep' the direction by using the contains switch.

**takes:** *dir*: directory of measurements (string) *contains*: will only load measurements who's filename contains this string.

**f\_unit:** frequency unit, to use for all networks. see `frequency.Frequency.unit` for info.

**returns:**

**ntwkDict:** a dictionary of calibrated measurements, the keys are the filenames.

**skrf.calibration.calibration.Calibration.biased\_error**

`Calibration.biased_error` (*std\_names*=None)

estimate of biased error for overdetermined calibration with multiple connections of each standard

**takes:**

**std\_names:** list of strings to uniquely identify each standard.\*

**returns:**

**systematic error:** `skrf.Network` type who's `.s_mag` is proportional to the systematic error metric

**note:**

mathematically, this is  $\text{mean}_s(|\text{mean}_c(\mathbf{r})|)$

**where:** *r*: complex residual errors *mean\_c*: complex mean taken accross connection *mean\_s*: complex mean taken accross standard

**skrf.calibration.calibration.Calibration.func\_per\_standard**

`Calibration.func_per_standard` (*func*, *attribute*='s', *std\_names*=None)

**skrf.calibration.calibration.Calibration.mean\_residuals**

`Calibration.mean_residuals` ()

**skrf.calibration.calibration.Calibration.plot\_coefs\_db**

`Calibration.plot_coefs_db` (*ax*=None, *show\_legend*=True, *\*\*kwargs*)

plot magnitude of the error coefficient dictionary

**skrf.calibration.calibration.Calibration.plot\_errors**

`Calibration.plot_errors` (*std\_names*=None, *scale*='db', *\*args*, *\*\*kwargs*)

plot calibration error metrics for an over-determined calibration.

see `biased_error`, `unbiased_error`, and `total_error` for more info

**skrf.calibration.calibration.Calibration.plot\_residuals**

`Calibration.plot_residuals` (*attribute*, \*args, \*\*kwargs)  
 plots a component of the residual errors on the Calibration-plane.

**takes:**

**attribute:** name of plotting method of Network class to call

**possible options are:** 'mag', 'db', 'smith', 'deg', etc

\*args,\*\*kwargs: passed to `plot_s_`'attribute'()

note: the residuals are calculated by:

`(self.apply_cal(self.measured[k])-self.ideals[k])`

**skrf.calibration.calibration.Calibration.plot\_residuals\_db**

`Calibration.plot_residuals_db` (\*args, \*\*kwargs)  
 see `plot_residuals`

**skrf.calibration.calibration.Calibration.plot\_residuals\_mag**

`Calibration.plot_residuals_mag` (\*args, \*\*kwargs)  
 see `plot_residuals`

**skrf.calibration.calibration.Calibration.plot\_residuals\_smith**

`Calibration.plot_residuals_smith` (\*args, \*\*kwargs)  
 see `plot_residuals`

**skrf.calibration.calibration.Calibration.plot\_uncertainty\_per\_standard**

`Calibration.plot_uncertainty_per_standard` (*scale='db'*, \*args, \*\*kwargs)  
 Plots uncertainty associated with each calibration standard.

This requires that each calibration standard is measured multiple times. The uncertainty associated with each standard is calculated by the complex standard deviation.

**Parameters** *scale*: 'db', 'lin'

plot uncertainties on linear or log scale

\*args, \*\*kwargs: passed to `uncertainty_per_standard()`

**See Also:**

`uncertainty_per_standard()`

**skrf.calibration.calibration.Calibration.run**

`Calibration.run()`  
 runs the calibration algorithm.

this is automatically called the first time any dependent property is referenced (like `error_ntwk`), but only the first time. if you change something and want to re-run the calibration

use this.

**skrf.calibration.calibration.Calibration.total\_error**

`Calibration.total_error` (*std\_names=None*)

estimate of total error for overdetermined calibration with multiple connections of each standard. This is the combined effects of both biased and un-biased errors

**takes:**

**std\_names:** list of strings to uniquely identify each standard.\*

**returns:**

**composit error:** skrf.Network type who's `.s_mag` is proportional to the composit error metric

**note:**

**mathematically, this is**  $\text{std\_cs}(r)$

**where:** `r`: complex residual errors `std_cs`: standard deviation taken accross connections and standards

**skrf.calibration.calibration.Calibration.unbiased\_error**

`Calibration.unbiased_error` (*std\_names=None*)

estimate of unbiased error for overdetermined calibration with multiple connections of each standard

**takes:**

**std\_names:** list of strings to uniquely identify each standard.\*

**returns:**

**stochastic error:** skrf.Network type who's `.s_mag` is proportional to the stochastic error metric

**see also:** `uncertainty_per_standard`, for this a measure of unbiased errors for each standard

**note:**

**mathematically, this is**  $\text{mean\_s}(\text{std\_c}(r))$

**where:** `r`: complex residual errors `std_c`: standard deviation taken accross connections `mean_s`: complex mean taken accross standards

**skrf.calibration.calibration.Calibration.uncertainty\_per\_standard**

`Calibration.uncertainty_per_standard` (*std\_names=None, attribute='s'*)

given that you have repeat-connections of single standard, this calculates the complex standard deviation (distance) for each standard in the calibration across connection #.

**takes:**

**std\_names:** list of strings to uniquely identify each standard.\*

**attribute:** string passed to `func_on_networks` to calculate std deviation on a component if desired. ['s']

**returns:** list of skrf.Networks, whose magnitude of s-parameters is proportional to the standard deviation for that standard

**\*example:**

**if your calibration had ideals named like:** 'short 1', 'short 2', 'open 1', 'open 2', etc.

**you would pass this** `mycal.uncertainty_per_standard(['short','open','match'])`



**skrf.calibration.calibration.Calibration.write**

`Calibration.write` (*file=None*, \*args, \*\*kwargs)

Write the Calibration to disk using `write()`

**Parameters** `file` : str or file-object

filename or a file-object. If left as None then the filename will be set to `Calibration.name`, if its not None. If both are None, `ValueError` is raised.

**\*args, \*\*kwargs** : arguments and keyword arguments

passed through to `write()`

**See Also:**

`skrf.io.general.write`, `skrf.io.general.read`

**Notes**

If the `self.name` is not None and `file` is can left as None and the resultant file will have the `.ntwk` extension appended to the filename.

**Examples**

```
>>> cal.name = 'my_cal'
>>> cal.write()
```

**calibrationAlgorithms (skrf.calibration.calibrationAlgorithms)**

Contains calibrations algorithms and related functions, which are used in the `Calibration` class.

**Calibration Algorithms**

<code>one_port(measured, ideals)</code>	Standard algorithm for a one port calibration.
<code>one_port_nls(measured, ideals)</code>	one port non-linear least squares.
<code>two_port(measured, ideals[, switch_terms])</code>	Two port calibration based on the 8-term error model.
<code>parameterized_self_calibration(measured, ideals)</code>	An iterative, general self-calibration routine.
<code>parameterized_self_calibration_nls(measured, ...)</code>	An iterative, general self-calibration routine.

**skrf.calibration.calibrationAlgorithms.one\_port**

`skrf.calibration.calibrationAlgorithms.one_port` (*measured, ideals*)

Standard algorithm for a one port calibration.

If more than three standards are supplied then a least square algorithm is applied.

**Parameters** `measured` : list of `Network` objects or `numpy.ndarray`

a list of the measured reflection coefficients. The elements of the list can either a `kxnxn` `numpy.ndarray`, representing a s-matrix, or list of 1-port `Network` objects.

`ideals` : list of `Network` objects or `numpy.ndarray`

a list of the ideal reflection coefficients. The elements of the list can either a `kxnxn` `numpy.ndarray`, representing a s-matrix, or list of 1-port `Network` objects.

**Returns** **output** : a dictionary

**output information from the calibration, the keys are**

- ‘error coefficients’: dictionary containing standard error coefficients
- ‘residuals’: a matrix of residuals from the least squared calculation. see `numpy.linalg.lstsq()` for more info

**See Also:**

`one_port_nls` for a non-linear least square implementation

### Notes

uses `numpy.linalg.lstsq()` for least squares calculation

### `skrf.calibration.calibrationAlgorithms.one_port_nls`

`skrf.calibration.calibrationAlgorithms.one_port_nls` (*measured*, *ideals*)  
one port non-linear least squares.

**Parameters** **measured** : list of `Network` objects or `numpy.ndarray`

a list of the measured reflection coefficients. The elements of the list can either a `kxnxn` `numpy.ndarray`, representing a s-matrix, or list of 1-port `Network` objects.

**ideals** : list of `Network` objects or `numpy.ndarray`

a list of the ideal reflection coefficients. The elements of the list can either a `kxnxn` `numpy.ndarray`, representing a s-matrix, or list of 1-port `Network` objects.

**Returns** **output** : a dictionary

a dictionary containing the following keys:

- ‘error coefficients’: dictionary containing standard error coefficients
- ‘residuals’: a matrix of residuals from the least squared calculation. see `numpy.linalg.lstsq()` for more info
- ‘cov\_x’: covariance matrix

### Notes

Uses `scipy.optimize.leastsq()` for non-linear least squares calculation

### `skrf.calibration.calibrationAlgorithms.two_port`

`skrf.calibration.calibrationAlgorithms.two_port` (*measured*, *ideals*,  
*switch\_terms=None*)

Two port calibration based on the 8-term error model.

Takes two ordered lists of measured and ideal responses. Optionally, switch terms<sup>33</sup> can be taken into account by passing a tuple containing the forward and reverse switch terms as 1-port `Network`s. This algorithm is based on the work in<sup>34</sup>.

---

<sup>33</sup> Marks, Roger B.; , “Formulations of the Basic Vector Network Analyzer Error Model including Switch-Terms,” ARFTG Conference Digest-Fall, 50th , vol.32, no., pp.115-126, Dec. 1997. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4119948&isnumber=4119931>

<sup>34</sup> Speciale, R.A.; , “A Generalization of the TSD Network-Analyzer Calibration Procedure, Covering n-Port Scattering-Parameter Measurements, Affected by Leakage Errors,” Microwave Theory and Techniques, IEEE Transactions on , vol.25, no.12, pp. 1100- 1115, Dec 1977. URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1129282&isnumber=25047>

**Parameters** **measured** : list of 2-port `Network` objects

Raw measurements of the calibration standards. The order must align with the *ideals* parameter

**ideals** : list of 2-port `Network` objects

Predicted ideal response of the calibration standards. The order must align with *ideals* list measured: ordered list of measured networks. list elements

**switch\_terms** : tuple of `Network` objects

The two measured switch terms in the order (forward, reverse). This is only applicable in two-port calibrations. See Roger Mark's paper on switch terms <sup>3</sup> for explanation of what they are.

**Returns** **output** : a dictionary

output information, contains the following keys: \* 'error coefficients': \* 'error vector': \* 'residuals':

## Notes

support for gathering switch terms on HP8510C is in `skrf.vi.vna`

## References

### `skrf.calibration.calibrationAlgorithms.parameterized_self_calibration`

`skrf.calibration.calibrationAlgorithms.parameterized_self_calibration` (*measured*, *ideals*, *showProgress=True*, *\*\*kwargs*)

An iterative, general self-calibration routine.

A self calibration routine based off of residual error minimization which can take any mixture of parameterized standards.

**Parameters** **measured** : list of `Network` objects

a list of the measured networks

**ideals** : list of `ParametricStandard` objects

a list of the ideal networks

**showProgress** : Boolean

turn printing progress on/off

**\*\*kwargs** : key-word arguments

passed to minimization algorithm (`scipy.optimize.fmin`)

**Returns** **output** : a dictionary

a dictionary containing the following keys:

- 'error\_coefficients' : dictionary of error coefficients
- 'residuals': residual matrix (shape depends on #stds)

- ‘parameter\_vector\_final’: final results for parameter vector
- **‘mean\_residual\_list’: the mean, magnitude of the residuals at each** iteration of calibration. this is the variable being minimized.

See Also:

**parametricStandard** sub-module for more info on them

**parameterized\_self\_calibration\_nls** similar algorithm, but uses a non-linear least-squares estimator

**skrf.calibration.calibrationAlgorithms.parameterized\_self\_calibration\_nls**

`skrf.calibration.calibrationAlgorithms.parameterized_self_calibration_nls` (*measured, ideals, als\_ps, showProgress=True, \*\*kwargs*)

An iterative, general self-calibration routine.

A self calibration routine based off of residual error minimization which can take any mixture of parameterized standards. Uses a non-linear least squares estimator to calculate the residuals.

**Parameters** **measured** : list of `Network` objects

a list of the measured networks

**ideals** : list of `Network` objects

a list of the ideal networks

**showProgress** : Boolean

turn printing progress on/off

**\*\*kwargs** : key-word arguments

passed to minimization algorithm (`scipy.optimize.fmin`)

**Returns** **output** : a dictionary

a dictionary containing the following keys:

- ‘error\_coefficients’ : dictionary of error coefficients
- ‘residuals’: residual matrix (shape depends on #stds)
- ‘parameter\_vector\_final’: final results for parameter vector
- **‘mean\_residual\_list’: the mean, magnitude of the residuals at each** iteration of calibration. this is the variable being minimized.

See Also:

**parametricStandard** sub-module for more info on them

**parameterized\_self\_calibration\_nls** similar algorithm, but uses a non-linear least-squares estimator

## Supporting Functions

---

<code>unterminate_switch_terms(two_port, gamma_f, ...)</code>	unterminates switch terms from raw measurements.
<code>abc_2_coefs_dict(abc)</code>	converts an abc ndarray to a dictionary containing the error
<code>eight_term_2_one_port_coefs(coefs)</code>	

---

**skrf.calibration.calibrationAlgorithms.unterminate\_switch\_terms**

`skrf.calibration.calibrationAlgorithms.unterminate_switch_terms` (*two\_port*,  
*gamma\_f*,  
*gamma\_r*)

unterminates switch terms from raw measurements.

**takes:** *two\_port*: the raw measurement, a 2-port Network type. *gamma\_f*: the measured forward switch term, a 1-port Network type *gamma\_r*: the measured reverse switch term, a 1-port Network type

**returns:** un-terminated measurement, a 2-port Network type

**see:** ‘Formulations of the Basic Vector Network Analyzer Error Model including Switch Terms’ by Roger B. Marks

**skrf.calibration.calibrationAlgorithms.abc\_2\_coefs\_dict**

`skrf.calibration.calibrationAlgorithms.abc_2_coefs_dict` (*abc*)  
converts an abc ndarray to a dictionary containing the error coefficients.

**takes:**

**abc** [Nx3 numpy.ndarray, which holds the complex calibration]

**coefficients. the components of abc are**  $a[:] = abc[:,0]$   $b[:] = abc[:,1]$   $c[:] = abc[:,2]$ ,

**a, b and c are related to the error network by**  $a = \det(e) = e_{01} * e_{10} - e_{00} * e_{11}$   $b = e_{00}$   $c = e_{11}$

**returns:**

**coefsDict: dictionary containing the following** ‘directivity’:*e00* ‘reflection tracking’:*e01e10* ‘source match’:*e11*

**note:** *e00* = directivity error *e10e01* = reflection tracking error *e11* = source match error

**skrf.calibration.calibrationAlgorithms.eight\_term\_2\_one\_port\_coefs**

`skrf.calibration.calibrationAlgorithms.eight_term_2_one_port_coefs` (*coefs*)

**calibrationFunctions (skrf.calibration.calibrationFunctions)**

Functions which operate on or pertain to `Calibration` Objects

---

<code>cartesian_product_calibration_set(ideals, ...)</code>	This function is used for calculating calibration uncertainty due to un-b
---	---

---

**skrf.calibration.calibrationFunctions.cartesian\_product\_calibration\_set**

`skrf.calibration.calibrationFunctions.cartesian_product_calibration_set` (*ideals*,  
*mea-*  
*sured*,  
*\*args*,  
*\*\*kwargs*)

This function is used for calculating calibration uncertainty due to un-biased, non-systematic errors.

It creates an ensemble of calibration instances. the set of measurement lists used in the ensemble is the Cartesian Product of all instances of each measured standard.

The idea is that if you have multiple measurements of each standard, then the multiple calibrations can be made by generating all possible combinations of measurements. This produces a conceptually simple, but computationally expensive way to estimate calibration uncertainty.

**takes:** ideals: list of ideal Networks measured: list of measured Networks \*args,\*\*kwargs: passed to Calibration initializer

**returns:** cal\_ensemble: a list of Calibration instances.

you can use the output to estimate uncertainty by calibrating a DUT with all calibrations, and then running statistics on the resultant set of Networks. for example

```
import skrf as rf # define you lists of ideals and measured networks
cal_ensemble = rf.cartesian_product_calibration_ensemble( ideals, measured)
dut = rf.Network('dut.s1p')
network_ensemble = [cal.apply_cal(dut) for cal in cal_ensemble]
rf.plot_uncertainty_mag(network_ensemble)
[network.plot_s_smith() for network in network_ensemble]
```

### 3.10.2 Classes

---

`Calibration(measured, ideals[, type, ...])` An object to represent a VNA calibration instance.

---

#### skrf.calibration.calibration.Calibration

**class** skrf.calibration.calibration.**Calibration** (*measured*, *ideals*, *type=None*, *is\_reciprocal=False*, *name=None*, *sloppy\_input=False*, *\*\*kwargs*)

An object to represent a VNA calibration instance.

A Calibration object is used to perform a calibration given a set measurements and ideals responses. It can run a calibration, store results, and apply the results to calculate corrected measurements.

#### Attributes

<code>Ts</code>	T-matrices used for de-embedding, a two-port calibration.
<code>caled_ntwk_sets</code>	returns a NetworkSet for each caled_ntwk, based on their names
<code>caled_ntwks</code>	list of the calibrated, calibration standards.
<code>calibration_algorithm_dict</code>	
<code>coefs</code>	coefs: a dictionary holding the calibration coefficients
<code>error_ntwk</code>	A Network object which represents the error network being
<code>nports</code>	the number of ports in the calibration
<code>nstandards</code>	number of ideal/measurement pairs in calibration
<code>output_from_cal</code>	a dictionary holding all of the output from the calibration
<code>residual_ntwks</code>	returns a the residuals for each calibration standard in the
<code>residuals</code>	if calibration is overdetermined, this holds the residuals
<code>type</code>	string representing what type of calibration is to be

**skrf.calibration.calibration.Calibration.Ts****Calibration.Ts**

T-matrices used for de-embedding, a two-port calibration.

**skrf.calibration.calibration.Calibration.caled\_ntwk\_sets****Calibration.caled\_ntwk\_sets**

returns a NetworkSet for each caled\_ntwk, based on their names

**skrf.calibration.calibration.Calibration.caled\_ntwks****Calibration.caled\_ntwks**

list of the calibrated, calibration standards.

**skrf.calibration.calibration.Calibration.calibration\_algorithm\_dict**

**Calibration.calibration\_algorithm\_dict** = {'two port': <function two\_port at 0x4b598c0>, 'one port parametric': dictionary holding calibration algorithms.

**skrf.calibration.calibration.Calibration.coefs****Calibration.coefs**

coefs: a dictionary holding the calibration coefficients

**for one port cal's** 'directivity':e00 'reflection tracking':e01e10 'source match':e11

**for 7-error term two port cal's** TODO:

**skrf.calibration.calibration.Calibration.error\_ntwk****Calibration.error\_ntwk**

A Network object which represents the error network being calibrated out.

**skrf.calibration.calibration.Calibration.nports****Calibration.nports**

the number of ports in the calibration

**skrf.calibration.calibration.Calibration.nstandards****Calibration.nstandards**

number of ideal/measurement pairs in calibration

**skrf.calibration.calibration.Calibration.output\_from\_cal****Calibration.output\_from\_cal**

a dictionary holding all of the output from the calibration algorithm

**skrf.calibration.calibration.Calibration.residual\_ntwks**

**Calibration.residual\_ntwks**

returns a the residuals for each calibration standard in the form of a list of Network types.

these residuals are calculated in the ‘calibrated domain’, meaning they are

$$r = (E.inv ** m - i)$$

**where**, r: residual network, E: embedding network, m: measured network i: ideal network

This way the units of the residual networks are meaningful

**note:** the residuals are only calculated if they are not existent.

so, if you want to re-calculate the residual networks then you delete the property ‘\_residual\_ntwks’.

**skrf.calibration.calibration.Calibration.residuals**

**Calibration.residuals**

if calibration is overdeteremined, this holds the residuals in the form of a vector.

also available are the complex residuals in the form of skrf.Network’s, see the property ‘residual\_ntwks’

**from numpy.lstsq:** residues: the sum of the residues; squared euclidean norm for each column vector in b (given ax=b)

**skrf.calibration.calibration.Calibration.type**

**Calibration.type**

string representing what type of calibration is to be performed. supported types at the moment are:

**‘one port’: standard one-port cal. if more than 2** measurement/ideal pairs are given it will calculate the least squares solution.

**‘two port’:** two port calibration based on the error-box model

**note:** algorithms referenced by calibration\_algorithm\_dict, are stored in calibrationAlgorithms.py

**Methods**

<code>__init__</code>	Calibration initializer.
<code>apply_cal</code>	apply the current calibration to a measurement.
<code>apply_cal_to_all_in_dir</code>	convience function to apply calibration to an entire directory
<code>biased_error</code>	estimate of biased error for overdetermined calibration with
<code>func_per_standard</code>	
<code>mean_residuals</code>	
<code>plot_coefs_db</code>	plot magnitude of the error coeficient dictionary
<code>plot_errors</code>	plot calibration error metrics for an over-determined calibration.
<code>plot_residuals</code>	plots a component of the residual errors on the Calibration-plane.
<code>plot_residuals_db</code>	see plot_residuals
<code>plot_residuals_mag</code>	see plot_residuals
<code>plot_residuals_smith</code>	see plot_residuals

Continued on next page



Table 3.41 – continued from previous page

<code>plot_uncertainty_per_standard</code>	Plots uncertainty associated with each calibration standard.
<code>run</code>	runs the calibration algorithm.
<code>total_error</code>	estimate of total error for overdetermined calibration with
<code>unbiased_error</code>	estimate of unbiased error for overdetermined calibration with
<code>uncertainty_per_standard</code>	given that you have repeat-connections of single standard,
<code>write</code>	Write the Calibration to disk using <code>write()</code>

**skrf.calibration.calibration.Calibration.\_\_init\_\_**

`Calibration.__init__(measured, ideals, type=None, is_reciprocal=False, name=None, sloppy_input=False, **kwargs)`

Calibration initializer.

**Parameters** `measured` : list of `Network` objects

Raw measurements of the calibration standards. The order must align with the `ideals` parameter

`ideals` : list of `Network` objects

Predicted ideal response of the calibration standards. The order must align with `ideals` list

**Notes**

All calibration algorithms are in stored in `skrf.calibration.calibrationAlgorithms`, refer to that file for documentation on the algorithms themselves. The Calibration class accesses those functions through the attribute `'calibration_algorithm_dict'`.

**References****Examples**

See the [Calibration](#) tutorial, or the examples sections for [One-Port Calibration](#) and [../../../../examples/twoport\\_calibration](#)

**skrf.calibration.calibration.Calibration.apply\_cal**

`Calibration.apply_cal(input_ntwk)`  
apply the current calibration to a measurement.

**takes:**

**input\_ntwk**: the measurement to apply the calibration to, a `Network` type.

**returns**: `caled`: the calibrated measurement, a `Network` type.

**skrf.calibration.calibration.Calibration.apply\_cal\_to\_all\_in\_dir**

`Calibration.apply_cal_to_all_in_dir(dir='.', contains=None, f_unit='ghz')`  
convenience function to apply calibration to an entire directory of measurements, and return a dictionary of the calibrated results, optionally the user can 'grep' the direction by using the `contains` switch.

**takes:** `dir`: directory of measurements (string) `contains`: will only load measurements who's filename contains this string.

**f\_unit:** frequency unit, to use for all networks. see `frequency.Frequency.unit` for info.

**returns:**

**ntwkDict:** a dictionary of calibrated measurements, the keys are the filenames.

#### `skrf.calibration.calibration.Calibration.biased_error`

`Calibration.biased_error` (*std\_names=None*)

estimate of biased error for overdetermined calibration with multiple connections of each standard

**takes:**

**std\_names:** list of strings to uniquely identify each standard.\*

**returns:**

**systematic error:** `skrf.Network` type who's `.s_mag` is proportional to the systematic error metric

**note:**

**mathematically, this is**  $\text{mean}_s(\text{lmean}_c(\mathbf{r}))$

**where:** `r`: complex residual errors `mean_c`: complex mean taken accross connection `mean_s`: complex mean taken accross standard

#### `skrf.calibration.calibration.Calibration.func_per_standard`

`Calibration.func_per_standard` (*func*, *attribute='s'*, *std\_names=None*)

#### `skrf.calibration.calibration.Calibration.mean_residuals`

`Calibration.mean_residuals` ()

#### `skrf.calibration.calibration.Calibration.plot_coefs_db`

`Calibration.plot_coefs_db` (*ax=None*, *show\_legend=True*, *\*\*kwargs*)

plot magnitude of the error coeficient dictionary

#### `skrf.calibration.calibration.Calibration.plot_errors`

`Calibration.plot_errors` (*std\_names=None*, *scale='db'*, *\*args*, *\*\*kwargs*)

plot calibration error metrics for an over-determined calibration.

see `biased_error`, `unbiased_error`, and `total_error` for more info

**skrf.calibration.calibration.Calibration.plot\_residuals**

`Calibration.plot_residuals` (*attribute*, \*args, \*\*kwargs)  
 plots a component of the residual errors on the Calibration-plane.

**takes:**

**attribute:** name of plotting method of Network class to call

**possible options are:** 'mag', 'db', 'smith', 'deg', etc

**\*args,\*\*kwargs:** passed to `plot_s_`'attribute'()

note: the residuals are calculated by:

`(self.apply_cal(self.measured[k])-self.ideals[k])`

**skrf.calibration.calibration.Calibration.plot\_residuals\_db**

`Calibration.plot_residuals_db` (\*args, \*\*kwargs)  
 see `plot_residuals`

**skrf.calibration.calibration.Calibration.plot\_residuals\_mag**

`Calibration.plot_residuals_mag` (\*args, \*\*kwargs)  
 see `plot_residuals`

**skrf.calibration.calibration.Calibration.plot\_residuals\_smith**

`Calibration.plot_residuals_smith` (\*args, \*\*kwargs)  
 see `plot_residuals`

**skrf.calibration.calibration.Calibration.plot\_uncertainty\_per\_standard**

`Calibration.plot_uncertainty_per_standard` (*scale='db'*, \*args, \*\*kwargs)  
 Plots uncertainty associated with each calibration standard.

This requires that each calibration standard is measured multiple times. The uncertainty associated with each standard is calculated by the complex standard deviation.

**Parameters** `scale` : 'db', 'lin'

plot uncertainties on linear or log scale

**\*args, \*\*kwargs** : passed to `uncertainty_per_standard()`

**See Also:**

`uncertainty_per_standard()`

**skrf.calibration.calibration.Calibration.run**

`Calibration.run` ()  
 runs the calibration algorithm.

this is automatically called the first time any dependent property is referenced (like `error_ntwk`), but only the first time. if you change something and want to re-run the calibration

use this.

#### `skrf.calibration.calibration.Calibration.total_error`

`Calibration.total_error` (*std\_names=None*)

estimate of total error for overdetermined calibration with multiple connections of each standard. This is the combined effects of both biased and un-biased errors

**takes:**

**std\_names:** list of strings to uniquely identify each standard.\*

**returns:**

**composit error:** `skrf.Network` type who's `.s_mag` is proportional to the composit error metric

**note:**

**mathematically, this is**  $\text{std\_cs}(r)$

**where:** `r`: complex residual errors `std_cs`: standard deviation taken accross connections and standards

#### `skrf.calibration.calibration.Calibration.unbiased_error`

`Calibration.unbiased_error` (*std\_names=None*)

estimate of unbiased error for overdetermined calibration with multiple connections of each standard

**takes:**

**std\_names:** list of strings to uniquely identify each standard.\*

**returns:**

**stochastic error:** `skrf.Network` type who's `.s_mag` is proportional to the stochastic error metric

**see also:** `uncertainty_per_standard`, for this a measure of unbiased errors for each standard

**note:**

**mathematically, this is**  $\text{mean\_s}(\text{std\_c}(r))$

**where:** `r`: complex residual errors `std_c`: standard deviation taken accross connections `mean_s`: complex mean taken accross standards

#### `skrf.calibration.calibration.Calibration.uncertainty_per_standard`

`Calibration.uncertainty_per_standard` (*std\_names=None, attribute='s'*)

given that you have repeat-connections of single standard, this calculates the complex standard deviation (distance) for each standard in the calibration across connection #.

**takes:**

**std\_names:** list of strings to uniquely identify each standard.\*

**attribute:** string passed to `func_on_networks` to calculate std deviation on a component if desired. ['s']

**returns:** list of `skrf.Networks`, whose magnitude of s-parameters is proportional to the standard deviation for that standard

**\*example:**

**if your calibration had ideals named like:** 'short 1', 'short 2', 'open 1', 'open 2', etc.  
**you would pass this** `mycal.uncertainty_per_standard(['short','open','match'])`

### `skrf.calibration.calibration.Calibration.write`

`Calibration.write` (*file=None*, \*args, \*\*kwargs)

Write the Calibration to disk using `write()`

**Parameters** `file` : str or file-object

filename or a file-object. If left as `None` then the filename will be set to `Calibration.name`, if its not `None`. If both are `None`, `ValueError` is raised.

**\*args, \*\*kwargs** : arguments and keyword arguments

passed through to `write()`

**See Also:**

`skrf.io.general.write`, `skrf.io.general.read`

### Notes

If the `self.name` is not `None` and `file` is can left as `None` and the resultant file will have the `.ntwk` extension appended to the filename.

### Examples

```
>>> cal.name = 'my_cal'
>>> cal.write()
```

## 3.11 media (`skrf.media`)

This package provides objects representing transmission line mediums.

The `Media` object is the base-class that is inherited by specific transmission line instances, such as `Freespace`, or `RectangularWaveguide`. The `Media` object provides generic methods to produce `Network`'s for any transmission line medium, such as `line()` and `delay_short()`. These methods are inherited by the specific transmission line classes, which internally define relevant quantities such as propagation constant, and characteristic impedance. This allows the specific transmission line mediums to produce networks without re-implementing methods for each specific media instance.

Network components specific to an given transmission line medium such as `cpw_short()` and `microstrip_bend()`, are implemented in those object

### 3.11.1 Media base-class

<code>Media</code>	The base-class for all transmission line mediums.
--------------------	---

### skrf.media.media.Media

**class** `skrf.media.media.Media` (*frequency*, *propagation\_constant*, *characteristic\_impedance*, *z0=None*)

The base-class for all transmission line mediums.

The `Media` object provides generic methods to produce `Network`'s for any transmission line medium, such as `line()` and `delay_short()`.

The initializer for this class has flexible argument types. This allows for the important attributes of the `Media` object to be dynamic. For example, if a `Media` object's propagation constant is a function of some attribute of that object, say *conductor\_width*, then the propagation constant will change when that attribute changes. See `__init__()` for details.

The network creation methods build off of each other. For example, the special load cases, such as `short()` and `open()` call `load()` with given arguments for `Gamma0`, and the `delay_` and `shunt_` functions call `line()` and `shunt()` respectively. This minimizes re-implementation.

Most methods initialize the `Network` by calling `match()` to create a 'blank' `Network`, and then fill in the s-matrix.

### Attributes

<code>characteristic_impedance</code>	Characterisitic impedance
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

### skrf.media.media.Media.characteristic\_impedance

`Media.characteristic_impedance`

Characterisitic impedance

The `characteristic_impedance` can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `characteristic_impedance` : `numpy.ndarray`

### skrf.media.media.Media.propagation\_constant

`Media.propagation_constant`

Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `propagation_constant` : `numpy.ndarray`

complex propagation constant for this media

## Notes

*propagation\_constant* must adhere to the following convention,

- positive real(propagation\_constant) = attenuation
- positive imag(propagation\_constant) = forward propagation

### skrf.media.media.Media.z0

#### Media.z0

Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is `None` then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `port_impedance` : `numpy.ndarray`  
the media's port impedance

## Methods

<code>__init__</code>	The Media initializer.
<code>capacitor</code>	Capacitor
<code>delay_load</code>	Delayed load
<code>delay_open</code>	Delayed open transmission line
<code>delay_short</code>	Delayed Short
<code>electrical_length</code>	calculates the electrical length for a given distance, at
<code>from_csv</code>	create a Media from numerical values stored in a csv file.
<code>guess_length_of_delay_short</code>	Guess physical length of a delay short.
<code>impedance_mismatch</code>	Two-port network for an impedance miss-match
<code>inductor</code>	Inductor
<code>line</code>	Matched transmission line of given length
<code>load</code>	Load of given reflection coefficient.
<code>match</code>	Perfect matched load ( $\Gamma_0 = 0$ ).
<code>open</code>	Open ( $\Gamma_0 = 1$ )
<code>resistor</code>	Resistor
<code>short</code>	Short ( $\Gamma_0 = -1$ )
<code>shunt</code>	Shunts a <code>Network</code>
<code>shunt_capacitor</code>	Shunted capacitor
<code>shunt_delay_load</code>	Shunted delayed load
<code>shunt_delay_open</code>	Shunted delayed open
<code>shunt_delay_short</code>	Shunted delayed short
<code>shunt_inductor</code>	Shunted inductor
<code>splitter</code>	Ideal, lossless n-way splitter.
<code>tee</code>	Ideal, lossless tee.

Continued on next page

**Table 3.44 – continued from previous page**

<code>theta_2_d</code>	Converts electrical length to physical distance.
<code>thru</code>	Matched transmission line of length 0.
<code>white_gaussian_polar</code>	Complex zero-mean gaussian white-noise network.
<code>write_csv</code>	write this media's frequency, $z_0$ , and gamma to a csv file.

**skrf.media.media.Media.\_\_init\_\_**

`Media.__init__(frequency, propagation_constant, characteristic_impedance, z0=None)`

The Media initializer.

This initializer has flexible argument types. The parameters *propagation\_constant*, *characteristic\_impedance* and *z0* can all be either static or dynamic. This is achieved by allowing those arguments to be either:

- functions which take no arguments or
- values (numbers or arrays)

In the case where the media's propagation constant may change after initialization, because you adjusted a parameter of the media, then passing the *propagation\_constant* as a function allows it to change when the media's parameters do.

**Parameters** `frequency` : `Frequency` object

frequency band of this transmission line medium

**propagation\_constant** : number, array-like, or a function

propagation constant for the medium.

**characteristic\_impedance** : number, array-like, or a function

characteristic impedance of transmission line medium.

**z0** : number, array-like, or a function

the port impedance for media, IF its different from the characteristic impedance of the transmission line medium (None) [a number]. if  $z_0 = \text{None}$  then will set to *characteristic\_impedance*

**See Also:**

`from_csv()` function to create a Media object from a csv file containing gamma/ $z_0$

**Notes**

***propagation\_constant* must adhere to the following convention,**

- positive `real(gamma)` = attenuation
- positive `imag(gamma)` = forward propagation

the  $z_0$  parameter is needed in some cases. For example, the `RectangularWaveguide` is an example where you may need this, because the characteristic impedance is frequency dependent, but the touchstone's created by most VNA's have  $z_0=1$



**skrf.media.media.Media.capacitor**`Media.capacitor` (*C*, **\*\*kwargs**)

Capacitor

**Parameters** *C* : number, array

Capacitance, in Farads. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** `capacitor` : a 2-port `Network`**See Also:**`match` function called to create a 'blank' network**skrf.media.media.Media.delay\_load**`Media.delay_load` (*Gamma0*, *d*, *unit='m'*, **\*\*kwargs**)

Delayed load

A load with reflection coefficient *Gamma0* at the end of a matched line of length *d*.**Parameters** *Gamma0* : number, array-like

reflection coefficient of load (not in dB)

**d** : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** `delay_load` : `Network` object

a delayed load

**See Also:**`line` creates the network for line`load` creates the network for the load**Notes**

This calls

```
line(d,unit, **kwargs) ** load(Gamma0, **kwargs)
```

### Examples

```
>>> my_media.delay_load(-.5, 90, 'deg', z0=50)
```

### skrf.media.media.Media.delay\_open

`Media.delay_open` (*d*, *unit*='m', *\*\*kwargs*)

Delayed open transmission line

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `delay_open` : `Network` object

a delayed open

**See Also:**

`delay_load` `delay_short` just calls this function

### skrf.media.media.Media.delay\_short

`Media.delay_short` (*d*, *unit*='m', *\*\*kwargs*)

Delayed Short

A transmission line of given length terminated with a short.

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `delay_short` : `Network` object

a delayed short

**See Also:**

`delay_load` `delay_short` just calls this function

### `skrf.media.media.Media.electrical_length`

`Media.electrical_length` (*d*, *deg=False*)

calculates the electrical length for a given distance, at the center frequency.

**Parameters** `d`: number or array-like :

delay distance, in meters

**deg**: Boolean :

return electrical length in deg?

**Returns** `theta`: number or array-like :

electrical length in radians or degrees, depending on value of `deg`.

### `skrf.media.media.Media.from_csv`

**classmethod** `Media.from_csv` (*filename*, *\*args*, *\*\*kwargs*)

create a `Media` from numerical values stored in a csv file.

the csv file format must be written by the function `write_csv()` which produces the following format

```
f[$unit], Re(Z0), Im(Z0), Re(gamma), Im(gamma), Re(port Z0), Im(port Z0) 1, 1, 1, 1, 1, 1 2, 1,
1, 1, 1, 1, 1 .....
```

### `skrf.media.media.Media.guess_length_of_delay_short`

`Media.guess_length_of_delay_short` (*aNtwk*)

Guess physical length of a delay short.

Unwraps the phase and determines the slope, which is then used in conjunction with `propagation_constant` to estimate the physical distance to the short.

**Parameters** `aNtwk` : `Network` object

(note: if this is a measurement it needs to be normalized to the reference plane)

### `skrf.media.media.Media.impedance_mismatch`

`Media.impedance_mismatch` (*z1*, *z2*, *\*\*kwargs*)

Two-port network for an impedance miss-match

**Parameters** `z1` : number, or array-like

complex impedance of port 1

`z2` : number, or array-like

complex impedance of port 2

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns mismatch** : `Network` object

a 2-port network representing the impedance mismatch

**See Also:**

`match` called to create a ‘blank’ network

**Notes**

If `z1` and `z2` are arrays, they must be of same length as the `Media.frequency.npoints`

### `skrf.media.media.Media.inductor`

`Media.inductor` (*L*, **\*\*kwargs**)

Inductor

**Parameters L** : number, array

Inductance, in Henrys. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns inductor** : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

### `skrf.media.media.Media.line`

`Media.line` (*d*, *unit='m'*, **\*\*kwargs**)

Matched transmission line of given length

The units of *length* are interpreted according to the value of *unit*.

**Parameters d** : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns line** : `Network` object

matched transmission line of given length

### Examples

```
>>> my_media.line(90, 'deg', z0=50)
```

### skrf.media.media.Media.load

`Media.load(Gamma0, nports=1, **kwargs)`

Load of given reflection coefficient.

**Parameters** **Gamma0** : number, array-like

Reflection coefficient of load (linear, not in db). If its an array it must be of shape:  $k \times n$ , where  $k$  is #frequency points in media, and  $n$  is *nports*

**nports** : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** **load** : class: '~skrf.network.Network' object :

n-port load, where  $S = \text{Gamma0} * \text{eye}(\dots)$

### skrf.media.media.Media.match

`Media.match(nports=1, z0=None, **kwargs)`

Perfect matched load ( $\Gamma_0 = 0$ ).

**Parameters** **nports** : int

number of ports

**z0** : number, or array-like

characteristic impedance. Default is None, in which case the Media's *z0* is used. This sets the resultant Network's *z0*.

**\*\*kwargs** : key word arguments

passed to `Network` initializer

**Returns** **match** : `Network` object

a n-port match

### Examples

```
>>> my_match = my_media.match(2, z0 = 50, name='Super Awesome Match')
```

**skrf.media.media.Media.open**

`Media.open` (*nports=1, \*\*kwargs*)

Open ( $\Gamma_0 = 1$ )

**Parameters** `nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `match` : `Network` object

a n-port open circuit

**See Also:**

`match` function called to create a ‘blank’ network

**skrf.media.media.Media.resistor**

`Media.resistor` (*R, \*args, \*\*kwargs*)

Resistor

**Parameters** `R` : number, array

Resistance, in Ohms. If this is an array, must be of same length as frequency vector.

**\*args, \*\*kwargs** : arguments, key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `resistor` : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

**skrf.media.media.Media.short**

`Media.short` (*nports=1, \*\*kwargs*)

Short ( $\Gamma_0 = -1$ )

**Parameters** `nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `match` : `Network` object

a n-port short circuit

**See Also:**

`match` function called to create a ‘blank’ network

**skrf.media.media.Media.shunt**

`Media.shunt` (*ntwk*, **\*\*kwargs**)

Shunts a `Network`

This creates a `tee()` and connects connects *ntwk* to port 1, and returns the result

**Parameters** `ntwk` : `Network` object

**\*\*kwargs** : keyword arguments

passed to `tee()`

**Returns** `shunted_ntwk` : `Network` object

a shunted a `ntwk`. The resultant `shunted_ntwk` will have  $(2 + \text{ntwk.number\_of\_ports} - 1)$  ports.

**skrf.media.media.Media.shunt\_capacitor**

`Media.shunt_capacitor` (*C*, *\*args*, **\*\*kwargs**)

Shunted capacitor

**Parameters** `C` : number, array-like

Capacitance in Farads.

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:`delay_open`

**Returns** `shunt_capacitor` : `Network` object

shunted capacitor(2-port)

**Notes**

This calls:

```
shunt(capacitor(C, *args, **kwargs))
```

**skrf.media.media.Media.shunt\_delay\_load**

`Media.shunt_delay_load` (*\*args*, **\*\*kwargs**)

Shunted delayed load

**Parameters** **\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:`delay_load`

**Returns** `shunt_delay_load` : `Network` object

a shunted delayed load (2-port)

### Notes

This calls:

```
shunt(delay_load(*args, **kwargs))
```

### skrf.media.media.Media.shunt\_delay\_open

Media.**shunt\_delay\_open**(\*args, \*\*kwargs)

Shunted delayed open

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_delay\_open** : *Network* object

shunted delayed open (2-port)

### Notes

This calls:

```
shunt(delay_open(*args, **kwargs))
```

### skrf.media.media.Media.shunt\_delay\_short

Media.**shunt\_delay\_short**(\*args, \*\*kwargs)

Shunted delayed short

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_delay\_load** : *Network* object

shunted delayed open (2-port)

### Notes

This calls:

```
shunt(delay_short(*args, **kwargs))
```

### skrf.media.media.Media.shunt\_inductor

Media.**shunt\_inductor**(L, \*args, \*\*kwargs)

Shunted inductor

**Parameters** L : number, array-like

Inductance in Farads.

\*args,\*\*kwargs : arguments, keyword arguments



passed to func:*delay\_open*

**Returns** `shunt_inductor` : `Network` object  
shunted inductor(2-port)

### Notes

This calls:

```
shunt(inductor(C, *args, **kwargs))
```

### `skrf.media.media.Media.splitter`

`Media.splitter` (*nports*, *\*\*kwargs*)  
Ideal, lossless n-way splitter.

**Parameters** `nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `tee` : `Network` object

a n-port splitter

**See Also:**

`match` called to create a ‘blank’ network

### `skrf.media.media.Media.tee`

`Media.tee` (*\*\*kwargs*)  
Ideal, lossless tee. (3-port splitter)

**Parameters** **\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `tee` : `Network` object

a 3-port splitter

**See Also:**

`splitter` this just calls `splitter(3)`

`match` called to create a ‘blank’ network

### `skrf.media.media.Media.theta_2_d`

`Media.theta_2_d` (*theta*, *deg=True*)  
Converts electrical length to physical distance.

The given electrical length is to be at the center frequency.

**Parameters** **theta** : number  
electrical length, at band center (see `deg` for unit)

**deg** : Boolean  
is theta in degrees?

**Returns** **d** : number  
physical distance in meters

#### `skrf.media.media.Media.thru`

`Media.thru(**kwargs)`  
Matched transmission line of length 0.

**Parameters** **\*\*kwargs** : key word arguments  
passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **thru** : `Network` object  
matched transmission line of 0 length

**See Also:**

`line` this just calls `line(0)`

#### `skrf.media.media.Media.white_gaussian_polar`

`Media.white_gaussian_polar(phase_dev, mag_dev, n_ports=1, **kwargs)`  
Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

**Parameters** **phase\_mag** : number  
standard deviation of magnitude

**phase\_dev** : number  
standard deviation of phase

**n\_ports** : int  
number of ports.

**\*\*kwargs** : passed to `Network`  
initializer

**Returns** **result** : `Network` object  
a noise network

**skrf.media.media.Media.write\_csv**

`Media.write_csv` (*filename*='f, gamma, z0.csv')  
write this media's frequency, z0, and gamma to a csv file.

**Parameters** `filename` : string  
file name to write out data to

**See Also:**

`from_csv` class method to initialize Media object from a csv file written from this function

**3.11.2 Transmission Line Classes**

<code>DistributedCircuit</code>	Generic, distributed circuit TEM transmission line
<code>RectangularWaveguide</code>	Rectangular Waveguide medium.
<code>CPW</code>	Coplanar waveguide class
<code>Freespace</code>	Represents a plane-wave in a homogeneous freespace, defined by the space's relative permativity and

**skrf.media.distributedCircuit.DistributedCircuit**

**class** `skrf.media.distributedCircuit.DistributedCircuit` (*frequency*, *C*, *I*, *R*, *G*, *\*args*,  
*\*\*kwargs*)

Generic, distributed circuit TEM transmission line

A TEM transmission line, defined in terms of distributed impedance and admittance values. A Distributed Circuit may be defined in terms of the following attributes,

Quantity	Symbol	Property
Distributed Capacitance	$C'$	C
Distributed Inductance	$I'$	I
Distributed Resistance	$R'$	R
Distributed Conductance	$G'$	G

From these, the following quantities may be calculated, which are functions of angular frequency ( $\omega$ ):

Quantity	Symbol	Property
Distributed Impedance	$Z' = R' + j\omega I'$	Z
Distributed Admittance	$Y' = G' + j\omega C'$	Y

From these we can calculate properties which define their wave behavior:

Quantity	Symbol	Method
Characteristic Impedance	$Z_0 = \sqrt{\frac{Z'}{Y'}}$	<code>z0 ()</code>
Propagation Constant	$\gamma = \sqrt{Z'Y'}$	<code>gamma ()</code>

Given the following definitions, the components of propagation constant are interpreted as follows:

$$+\Re\{\gamma\} = \text{attenuation}$$

$$-\Im\{\gamma\} = \text{forward propagation}$$

**Attributes**

<code>Y</code>	Distributed Admittance, $Y'$
<code>Z</code>	Distributed Impedance, $Z'$
<code>characteristic_impedance</code>	Characterisitic impedance
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

#### `skrf.media.distributedCircuit.DistributedCircuit.Y`

`DistributedCircuit.Y`  
Distributed Admittance,  $Y'$

Defined as

$$Y' = G' + j\omega C'$$

**Returns** `Y` : `numpy.ndarray`

Distributed Admittance in units of S/m

#### `skrf.media.distributedCircuit.DistributedCircuit.Z`

`DistributedCircuit.Z`  
Distributed Impedance,  $Z'$

Defined as

$$Z' = R' + j\omega L'$$

**Returns** `Z` : `numpy.ndarray`

Distributed impedance in units of ohm/m

#### `skrf.media.distributedCircuit.DistributedCircuit.characteristic_impedance`

`DistributedCircuit.characteristic_impedance`  
Characterisitic impedance

The `characteristic_impedance` can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `characteristic_impedance` : `numpy.ndarray`

#### `skrf.media.distributedCircuit.DistributedCircuit.propagation_constant`

`DistributedCircuit.propagation_constant`  
Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `propagation_constant` : `numpy.ndarray`  
 complex propagation constant for this media

## Notes

*propagation\_constant* must adhere to the following convention,

- positive real(propagation\_constant) = attenuation
- positive imag(propagation\_constant) = forward propagation

## skrf.media.distributedCircuit.DistributedCircuit.z0

`DistributedCircuit.z0`

Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is `None` then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `port_impedance` : `numpy.ndarray`  
 the media's port impedance

## Methods

<code>z0</code>	Characteristic Impedance, $Z_0$
<code>__init__</code>	Distributed Circuit constructor.
<code>capacitor</code>	Capacitor
<code>delay_load</code>	Delayed load
<code>delay_open</code>	Delayed open transmission line
<code>delay_short</code>	Delayed Short
<code>electrical_length</code>	calculates the electrical length for a given distance, at
<code>from_Media</code>	Initializes a DistributedCircuit from an existing
<code>from_csv</code>	create a Media from numerical values stored in a csv file.
<code>gamma</code>	Propagation Constant, $\gamma$
<code>guess_length_of_delay_short</code>	Guess physical length of a delay short.
<code>impedance_mismatch</code>	Two-port network for an impedance miss-match
<code>inductor</code>	Inductor
<code>line</code>	Matched transmission line of given length
<code>load</code>	Load of given reflection coefficient.
<code>match</code>	Perfect matched load ( $\Gamma_0 = 0$ ).
<code>open</code>	Open ( $\Gamma_0 = 1$ )
<code>resistor</code>	Resistor
<code>short</code>	Short ( $\Gamma_0 = -1$ )
<code>shunt</code>	Shunts a <code>Network</code>

Continued on next page

**Table 3.47 – continued from previous page**

<code>shunt_capacitor</code>	Shunted capacitor
<code>shunt_delay_load</code>	Shunted delayed load
<code>shunt_delay_open</code>	Shunted delayed open
<code>shunt_delay_short</code>	Shunted delayed short
<code>shunt_inductor</code>	Shunted inductor
<code>splitter</code>	Ideal, lossless n-way splitter.
<code>tee</code>	Ideal, lossless tee.
<code>theta_2_d</code>	Converts electrical length to physical distance.
<code>thru</code>	Matched transmission line of length 0.
<code>white_gaussian_polar</code>	Complex zero-mean gaussian white-noise network.
<code>write_csv</code>	write this media's frequency, $z_0$ , and gamma to a csv file.

**skrf.media.distributedCircuit.DistributedCircuit.Z0**

`DistributedCircuit.Z0()`  
Characteristic Impedance,  $Z_0$

$$Z_0 = \sqrt{\frac{Z'}{Y'}}$$

**Returns** **Z0** : numpy.ndarray  
Characteristic Impedance in units of ohms

**skrf.media.distributedCircuit.DistributedCircuit.\_\_init\_\_**

`DistributedCircuit.__init__(frequency, C, I, R, G, *args, **kwargs)`  
Distributed Circuit constructor.

**Parameters** **frequency** : `Frequency` object

- C** : number, or array-like  
distributed capacitance, in F/m
- I** : number, or array-like  
distributed inductance, in H/m
- R** : number, or array-like  
distributed resistance, in Ohm/m
- G** : number, or array-like  
distributed conductance, in S/m

**Notes**

C,I,R,G can all be vectors as long as they are the same length

This object can be constructed from a `Media` instance too, see the classmethod `from_Media()`

**skrf.media.distributedCircuit.DistributedCircuit.capacitor**

DistributedCircuit.**capacitor**(*C*, **\*\*kwargs**)

Capacitor

**Parameters** *C* : number, array

Capacitance, in Farads. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **capacitor** : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

**skrf.media.distributedCircuit.DistributedCircuit.delay\_load**

DistributedCircuit.**delay\_load**(*Gamma0*, *d*, *unit='m'*, **\*\*kwargs**)

Delayed load

A load with reflection coefficient *Gamma0* at the end of a matched line of length *d*.

**Parameters** **Gamma0** : number, array-like

reflection coefficient of load (not in dB)

**d** : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **delay\_load** : `Network` object

a delayed load

**See Also:**

`line` creates the network for line

`load` creates the network for the load

**Notes**

This calls

```
line(d,unit, **kwargs) ** load(Gamma0, **kwargs)
```

### Examples

```
>>> my_media.delay_load(-.5, 90, 'deg', z0=50)
```

### skrf.media.distributedCircuit.DistributedCircuit.delay\_open

DistributedCircuit.**delay\_open** (*d*, *unit*='m', *\*\*kwargs*)

Delayed open transmission line

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** **delay\_object** : `Network` object

a delayed open

**See Also:**

**delay\_load** `delay_short` just calls this function

### skrf.media.distributedCircuit.DistributedCircuit.delay\_short

DistributedCircuit.**delay\_short** (*d*, *unit*='m', *\*\*kwargs*)

Delayed Short

A transmission line of given length terminated with a short.

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.



**Returns** `delay_short` : `Network` object

a delayed short

**See Also:**

`delay_load` `delay_short` just calls this function

### `skrf.media.distributedCircuit.DistributedCircuit.electrical_length`

`DistributedCircuit.electrical_length` (*d*, *deg=False*)

calculates the electrical length for a given distance, at the center frequency.

**Parameters** *d*: number or array-like :

delay distance, in meters

**deg**: Boolean :

return electral length in deg?

**Returns** *theta*: number or array-like :

electrical length in radians or degrees, depending on value of *deg*.

### `skrf.media.distributedCircuit.DistributedCircuit.from_Media`

**classmethod** `DistributedCircuit.from_Media` (*my\_media*, *\*args*, *\*\*kwargs*)

Initializes a `DistributedCircuit` from an existing `:class:'~skrf.media.media.Media'` instance.

### `skrf.media.distributedCircuit.DistributedCircuit.from_csv`

**classmethod** `DistributedCircuit.from_csv` (*filename*, *\*args*, *\*\*kwargs*)

create a `Media` from numerical values stored in a csv file.

the csv file format must be written by the function `write_csv()` which produces the following format

```
f[$unit], Re(Z0), Im(Z0), Re(gamma), Im(gamma), Re(port Z0), Im(port Z0) 1, 1, 1, 1, 1, 1, 1 2, 1,
1, 1, 1, 1, 1 .....
```

### `skrf.media.distributedCircuit.DistributedCircuit.gamma`

`DistributedCircuit.gamma` ()

Propagation Constant,  $\gamma$

Defined as,

$$\gamma = \sqrt{Z'Y'}$$

**Returns** `gamma` : `numpy.ndarray`

Propagation Constant,

## Notes

The components of propagation constant are interpreted as follows:  
positive real( $\gamma$ ) = attenuation positive imag( $\gamma$ ) = forward propagation

### `skrf.media.distributedCircuit.DistributedCircuit.guess_length_of_delay_short`

`DistributedCircuit.guess_length_of_delay_short` (*aNtwk*)  
Guess physical length of a delay short.

Unwraps the phase and determines the slope, which is then used in conjunction with `propagation_constant` to estimate the physical distance to the short.

**Parameters** `aNtwk` : `Network` object

(note: if this is a measurement it needs to be normalized to the reference plane)

### `skrf.media.distributedCircuit.DistributedCircuit.impedance_mismatch`

`DistributedCircuit.impedance_mismatch` (*z1*, *z2*, *\*\*kwargs*)  
Two-port network for an impedance miss-match

**Parameters** `z1` : number, or array-like

complex impedance of port 1

`z2` : number, or array-like

complex impedance of port 2

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `missmatch` : `Network` object

a 2-port network representing the impedance mismatch

## See Also:

`match` called to create a 'blank' network

## Notes

If `z1` and `z2` are arrays, they must be of same length as the `Media.frequency.npoints`

### `skrf.media.distributedCircuit.DistributedCircuit.inductor`

`DistributedCircuit.inductor` (*L*, *\*\*kwargs*)  
Inductor

**Parameters** `L` : number, array

Inductance, in Henrys. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `inductor` : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

### `skrf.media.distributedCircuit.DistributedCircuit.line`

`DistributedCircuit.line` (*d*, *unit*='m', *\*\*kwargs*)

Matched transmission line of given length

The units of *length* are interpreted according to the value of *unit*.

**Parameters** `d` : number

the length of transmissin line (see unit argument)

`unit` : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `line` : `Network` object

matched tranmission line of given length

### Examples

```
>>> my_media.line(90, 'deg', z0=50)
```

### `skrf.media.distributedCircuit.DistributedCircuit.load`

`DistributedCircuit.load` (*Gamma0*, *nports*=1, *\*\*kwargs*)

Load of given reflection coefficient.

**Parameters** `Gamma0` : number, array-like

Reflection coefficient of load (linear, not in db). If its an array it must be of shape:  $k \times n$ , where  $k$  is #frequency points in media, and  $n$  is *nports*

`nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `load` :class:‘~skrf.network.Network’ object :

n-port load, where  $S = \text{Gamma0} * \text{eye}(\dots)$

**skrf.media.distributedCircuit.DistributedCircuit.match**

DistributedCircuit.**match** (*nports=1, z0=None, \*\*kwargs*)

Perfect matched load ( $\Gamma_0 = 0$ ).

**Parameters** **nports** : int

number of ports

**z0** : number, or array-like

characteristic impedance. Default is None, in which case the Media's *z0* is used. This sets the resultant Network's *z0*.

**\*\*kwargs** : key word arguments

passed to `Network` initializer

**Returns** **match** : `Network` object

a n-port match

**Examples**

```
>>> my_match = my_media.match(2, z0 = 50, name='Super Awesome Match')
```

**skrf.media.distributedCircuit.DistributedCircuit.open**

DistributedCircuit.**open** (*nports=1, \*\*kwargs*)

Open ( $\Gamma_0 = 1$ )

**Parameters** **nports** : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** **match** : `Network` object

a n-port open circuit

**See Also:**

`match` function called to create a 'blank' network

**skrf.media.distributedCircuit.DistributedCircuit.resistor**

DistributedCircuit.**resistor** (*R, \*args, \*\*kwargs*)

Resistor

**Parameters** **R** : number, array

Resistance, in Ohms. If this is an array, must be of same length as frequency vector.

**\*args, \*\*kwargs** : arguments, key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns resistor** : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

#### `skrf.media.distributedCircuit.DistributedCircuit.short`

`DistributedCircuit.short` (*nports=1, \*\*kwargs*)

Short ( $\Gamma_0 = -1$ )

**Parameters nports** : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns match** : `Network` object

a n-port short circuit

**See Also:**

`match` function called to create a ‘blank’ network

#### `skrf.media.distributedCircuit.DistributedCircuit.shunt`

`DistributedCircuit.shunt` (*ntwk, \*\*kwargs*)

Shunts a `Network`

This creates a `tee()` and connects connects *ntwk* to port 1, and returns the result

**Parameters ntwk** : `Network` object

**\*\*kwargs** : keyword arguments

passed to `tee()`

**Returns shunted\_ntwk** : `Network` object

a shunted a ntwk. The resultant shunted\_ntwk will have  $(2 + \text{ntwk.number\_of\_ports} - 1)$  ports.

#### `skrf.media.distributedCircuit.DistributedCircuit.shunt_capacitor`

`DistributedCircuit.shunt_capacitor` (*C, \*args, \*\*kwargs*)

Shunted capacitor

**Parameters C** : number, array-like

Capacitance in Farads.

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:*delay\_open*

**Returns shunt\_capacitor** : `Network` object

shunted capacitor(2-port)

### Notes

This calls:

```
shunt(capacitor(C,*args, **kwargs))
```

### skrf.media.distributedCircuit.DistributedCircuit.shunt\_delay\_load

DistributedCircuit.**shunt\_delay\_load**(\*args, \*\*kwargs)

Shunted delayed load

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_load*

**Returns** **shunt\_delay\_load** : *Network* object

a shunted delayed load (2-port)

### Notes

This calls:

```
shunt(delay_load(*args, **kwargs))
```

### skrf.media.distributedCircuit.DistributedCircuit.shunt\_delay\_open

DistributedCircuit.**shunt\_delay\_open**(\*args, \*\*kwargs)

Shunted delayed open

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_delay\_open** : *Network* object

shunted delayed open (2-port)

### Notes

This calls:

```
shunt(delay_open(*args, **kwargs))
```

### skrf.media.distributedCircuit.DistributedCircuit.shunt\_delay\_short

DistributedCircuit.**shunt\_delay\_short**(\*args, \*\*kwargs)

Shunted delayed short

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_delay\_load** : *Network* object

shunted delayed open (2-port)

**Notes**

This calls:

```
shunt(delay_short(*args, **kwargs))
```

**skrf.media.distributedCircuit.DistributedCircuit.shunt\_inductor**

DistributedCircuit.**shunt\_inductor**(*L*, \*args, \*\*kwargs)

Shunted inductor

**Parameters** *L* : number, array-like

Inductance in Farads.

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_inductor** : *Network* object

shunted inductor(2-port)

**Notes**

This calls:

```
shunt(inductor(C, *args, **kwargs))
```

**skrf.media.distributedCircuit.DistributedCircuit.splitter**

DistributedCircuit.**splitter**(*nports*, \*\*kwargs)

Ideal, lossless n-way splitter.

**Parameters** *nports* : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **tee** : *Network* object

a n-port splitter

**See Also:**

`match` called to create a ‘blank’ network

**skrf.media.distributedCircuit.DistributedCircuit.tee**

DistributedCircuit.**tee**(\*\*kwargs)

Ideal, lossless tee. (3-port splitter)

**Parameters** **\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `tee`: `Network` object  
a 3-port splitter

**See Also:**

`splitter` this just calls `splitter(3)`  
`match` called to create a ‘blank’ network

#### `skrf.media.distributedCircuit.DistributedCircuit.theta_2_d`

`DistributedCircuit.theta_2_d(theta, deg=True)`  
Converts electrical length to physical distance.

The given electrical length is to be at the center frequency.

**Parameters** `theta`: number  
electrical length, at band center (see `deg` for unit)  
`deg`: Boolean  
is theta in degrees?

**Returns** `d`: number  
physical distance in meters

#### `skrf.media.distributedCircuit.DistributedCircuit.thru`

`DistributedCircuit.thru(**kwargs)`  
Matched transmission line of length 0.

**Parameters** `**kwargs`: key word arguments  
passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `thru`: `Network` object  
matched transmission line of 0 length

**See Also:**

`line` this just calls `line(0)`

#### `skrf.media.distributedCircuit.DistributedCircuit.white_gaussian_polar`

`DistributedCircuit.white_gaussian_polar(phase_dev, mag_dev, n_ports=1, **kwargs)`  
Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

**Parameters** `phase_mag`: number  
standard deviation of magnitude  
`phase_dev`: number  
standard deviation of phase



**n\_ports** : int  
 number of ports.  
**\*\*kwargs** : passed to `Network`  
 initializer

**Returns result** : `Network` object  
 a noise network

**skrf.media.distributedCircuit.DistributedCircuit.write\_csv**

`DistributedCircuit.write_csv` (*filename='f, gamma, z0.csv'*)  
 write this media's frequency, z0, and gamma to a csv file.

**Parameters filename** : string  
 file name to write out data to

**See Also:**

`from_csv` class method to initialize Media object from a csv file written from this function

**skrf.media.rectangularWaveguide.RectangularWaveguide**

**class** `skrf.media.rectangularWaveguide.RectangularWaveguide` (*frequency, a, b=None, mode\_type='te', m=1, n=0, ep\_r=1, mu\_r=1, \*args, \*\*kwargs*)

Rectangular Waveguide medium.

Represents a single mode of a homogeneously filled rectangular waveguide of cross-section *a* x *b*. The mode is determined by mode-type (te or tm) and mode indices ( *m* and *n* ).

Quantity	Symbol	Variable
Characteristic Wave Number	$k_0$	k0
Cut-off Wave Number	$k_c$	kc
Longitudinal Wave Number	$k_z$	kz
Transverse Wave Number (a)	$k_x$	kx
Transverse Wave Number (b)	$k_y$	ky
Characteristic Impedance	$Z_0$	z0

**Attributes**

<code>characteristic_impedance</code>	Characterisitic impedance
<code>ep</code>	The permativity of the filling material
<code>k0</code>	Characteristic wave number
<code>kc</code>	Cut-off wave number
<code>kx</code>	Eigen value in the 'a' direction
<code>ky</code>	Eigen-value in the 'b' direction.
<code>mu</code>	The permeability of the filling material
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

**skrf.media.rectangularWaveguide.RectangularWaveguide.characteristic\_impedance**

RectangularWaveguide.**characteristic\_impedance**

Characterisitic impedance

The characteristic\_impedance can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** **characteristic\_impedance** : `numpy.ndarray`

**skrf.media.rectangularWaveguide.RectangularWaveguide.ep**

RectangularWaveguide.**ep**

The permativity of the filling material

**Returns** **ep** : number

filling material's relative permativity

**skrf.media.rectangularWaveguide.RectangularWaveguide.k0**

RectangularWaveguide.**k0**

Characteristic wave number

**Returns** **k0** : number

characteristic wave number

**skrf.media.rectangularWaveguide.RectangularWaveguide.kc**

RectangularWaveguide.**kc**

Cut-off wave number

Defined as

$$k_c = \sqrt{k_x^2 + k_y^2} = \sqrt{m \frac{\pi^2}{a} + n \frac{\pi^2}{b}}$$

**Returns** **kc** : number

cut-off wavenumber

**skrf.media.rectangularWaveguide.RectangularWaveguide.kx**

RectangularWaveguide.**kx**

Eigen value in the 'a' direction

Defined as

$$k_x = m \frac{\pi}{a}$$

**Returns** **kx** : number

eigen-value in *a* direction

**skrf.media.rectangularWaveguide.RectangularWaveguide.ky**

`RectangularWaveguide.ky`  
Eigen-value in the  $b$  direction.

Defined as

$$k_y = n \frac{\pi}{b}$$

**Returns** `ky` : number  
eigen-value in  $b$  direction

**skrf.media.rectangularWaveguide.RectangularWaveguide.mu**

`RectangularWaveguide.mu`  
The permeability of the filling material

**Returns** `mu` : number  
filling material's relative permeability

**skrf.media.rectangularWaveguide.RectangularWaveguide.propagation\_constant**

`RectangularWaveguide.propagation_constant`  
Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `propagation_constant` : `numpy.ndarray`  
complex propagation constant for this media

**Notes**

***propagation\_constant* must adhere to the following convention,**

- positive `real(propagation_constant)` = attenuation
- positive `imag(propagation_constant)` = forward propagation

**skrf.media.rectangularWaveguide.RectangularWaveguide.z0**

`RectangularWaveguide.z0`  
Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is `None` then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `port_impedance` : `numpy.ndarray`  
the media's port impedance

### Methods

<code>z0</code>	The characteristic impedance
<code>__init__</code>	RectangularWaveguide initializer
<code>capacitor</code>	Capacitor
<code>delay_load</code>	Delayed load
<code>delay_open</code>	Delayed open transmission line
<code>delay_short</code>	Delayed Short
<code>electrical_length</code>	calculates the electrical length for a given distance, at
<code>from_csv</code>	create a Media from numerical values stored in a csv file.
<code>guess_length_of_delay_short</code>	Guess physical length of a delay short.
<code>impedance_mismatch</code>	Two-port network for an impedance miss-match
<code>inductor</code>	Inductor
<code>kz</code>	The Longitudinal wave number, aka propagation constant.
<code>line</code>	Matched transmission line of given length
<code>load</code>	Load of given reflection coefficient.
<code>match</code>	Perfect matched load ( $\Gamma_0 = 0$ ).
<code>open</code>	Open ( $\Gamma_0 = 1$ )
<code>resistor</code>	Resistor
<code>short</code>	Short ( $\Gamma_0 = -1$ )
<code>shunt</code>	Shunts a <code>Network</code>
<code>shunt_capacitor</code>	Shunted capacitor
<code>shunt_delay_load</code>	Shunted delayed load
<code>shunt_delay_open</code>	Shunted delayed open
<code>shunt_delay_short</code>	Shunted delayed short
<code>shunt_inductor</code>	Shunted inductor
<code>splitter</code>	Ideal, lossless n-way splitter.
<code>tee</code>	Ideal, lossless tee.
<code>theta_2_d</code>	Converts electrical length to physical distance.
<code>thru</code>	Matched transmission line of length 0.
<code>white_gaussian_polar</code>	Complex zero-mean gaussian white-noise network.
<code>write_csv</code>	write this media's frequency, z0, and gamma to a csv file.

### `skrf.media.rectangularWaveguide.RectangularWaveguide.z0`

`RectangularWaveguide.z0()`  
The characteristic impedance

### `skrf.media.rectangularWaveguide.RectangularWaveguide.__init__`

`RectangularWaveguide.__init__(frequency, a, b=None, mode_type='te', m=1, n=0, ep_r=1, mu_r=1, *args, **kwargs)`  
RectangularWaveguide initializer

**Parameters** `frequency` : class:~`skrf.frequency.Frequency` object  
frequency band for this media

**a** : number  
width of waveguide, in meters.

**b** : number  
height of waveguide, in meters. If *None* defaults to  $a/2$

**mode\_type** : ['te','tm']  
mode type, transverse electric (te) or transverse magnetic (tm) to-z. where z is direction of propagation

**m** : int  
mode index in 'a'-direction

**n** : int  
mode index in 'b'-direction

**ep\_r** : number, array-like,  
filling material's relative permativity

**mu\_r** : number, array-like  
filling material's relative permeability

**\*args,\*\*kwargs** : arguments, keyword arguments  
passed to `Media`'s constructor (`__init__()`)

### Examples

Most common usage is standard aspect ratio (2:1) dominant mode, TE10 mode of wr10 waveguide can be constructed by

```
>>> freq = rf.Frequency(75,110,101,'ghz')
>>> rf.RectangularWaveguide(freq, 100*mil)
```

### `skrf.media.rectangularWaveguide.RectangularWaveguide.capacitor`

`RectangularWaveguide.capacitor` (*C*, *\*\*kwargs*)  
Capacitor

**Parameters** *C* : number, array

Capacitance, in Farads. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `capacitor` : a 2-port `Network`

**See Also:**

`match` function called to create a 'blank' network

**skrf.media.rectangularWaveguide.RectangularWaveguide.delay\_load**`RectangularWaveguide.delay_load(Gamma0, d, unit='m', **kwargs)`

Delayed load

A load with reflection coefficient  $\Gamma$  at the end of a matched line of length  $d$ .**Parameters** **Gamma0** : number, array-like

reflection coefficient of load (not in dB)

**d** : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **delay\_load** : `Network` object

a delayed load

**See Also:****line** creates the network for line**load** creates the network for the load**Notes**

This calls

```
line(d,unit, **kwargs) ** load(Gamma0, **kwargs)
```

**Examples**

```
>>> my_media.delay_load(-.5, 90, 'deg', z0=50)
```

**skrf.media.rectangularWaveguide.RectangularWaveguide.delay\_open**`RectangularWaveguide.delay_open(d, unit='m', **kwargs)`

Delayed open transmission line

**Parameters** **d** : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `delay_open` : `Network` object

a delayed open

**See Also:**

`delay_load` `delay_short` just calls this function

### `skrf.media.rectangularWaveguide.RectangularWaveguide.delay_short`

`RectangularWaveguide.delay_short` (*d*, *unit*='m', **\*\*kwargs**)  
Delayed Short

A transmission line of given length terminated with a short.

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `delay_short` : `Network` object

a delayed short

**See Also:**

`delay_load` `delay_short` just calls this function

### `skrf.media.rectangularWaveguide.RectangularWaveguide.electrical_length`

`RectangularWaveguide.electrical_length` (*d*, *deg*=False)  
calculates the electrical length for a given distance, at the center frequency.

**Parameters** *d*: number or array-like :

delay distance, in meters

**deg**: Boolean :

return electral length in deg?

**Returns** *theta*: number or array-like :

electrical length in radians or degrees, depending on value of deg.

#### `skrf.media.rectangularWaveguide.RectangularWaveguide.from_csv`

**classmethod** `RectangularWaveguide.from_csv` (*filename*, \**args*, \*\**kwargs*)

create a Media from numerical values stored in a csv file.

the csv file format must be written by the function `write_csv()` which produces the following format

```
f[$unit], Re(Z0), Im(Z0), Re(gamma), Im(gamma), Re(port Z0), Im(port Z0) 1, 1, 1, 1, 1, 1 2, 1,
1, 1, 1, 1, 1 .....
```

#### `skrf.media.rectangularWaveguide.RectangularWaveguide.guess_length_of_delay_short`

`RectangularWaveguide.guess_length_of_delay_short` (*aNtwk*)

Guess physical length of a delay short.

Unwraps the phase and determines the slope, which is then used in conjunction with `propagation_constant` to estimate the physical distance to the short.

**Parameters** `aNtwk` : `Network` object

(note: if this is a measurement it needs to be normalized to the reference plane)

#### `skrf.media.rectangularWaveguide.RectangularWaveguide.impedance_mismatch`

`RectangularWaveguide.impedance_mismatch` (*z1*, *z2*, \*\**kwargs*)

Two-port network for an impedance miss-match

**Parameters** `z1` : number, or array-like

complex impedance of port 1

`z2` : number, or array-like

complex impedance of port 2

\*\*`kwargs` : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `missmatch` : `Network` object

a 2-port network representing the impedance mismatch

#### See Also:

`match` called to create a 'blank' network

#### Notes

If `z1` and `z2` are arrays, they must be of same length as the `Media.frequency.npoints`



**skrf.media.rectangularWaveguide.RectangularWaveguide.inductor**

RectangularWaveguide.**inductor** (*L*, **\*\*kwargs**)

Inductor

**Parameters** *L* : number, array

Inductance, in Henrys. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **inductor** : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

**skrf.media.rectangularWaveguide.RectangularWaveguide.kz**

RectangularWaveguide.**kz** ()

The Longitudinal wave number, aka propagation constant.

Defined as

$$k_z = \pm \sqrt{k_0^2 - k_c^2}$$

**This is.**

- IMAGINARY for propagating modes
- REAL for non-propagating modes,

**Returns** **kz** : number

The propagation constant

**skrf.media.rectangularWaveguide.RectangularWaveguide.line**

RectangularWaveguide.**line** (*d*, *unit='m'*, **\*\*kwargs**)

Matched transmission line of given length

The units of *length* are interpreted according to the value of *unit*.

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `line` : `Network` object  
matched transmission line of given length

### Examples

```
>>> my_media.line(90, 'deg', z0=50)
```

### `skrf.media.rectangularWaveguide.RectangularWaveguide.load`

`RectangularWaveguide.load` (*Gamma0*, *nports=1*, *\*\*kwargs*)

Load of given reflection coefficient.

**Parameters** `Gamma0` : number, array-like

Reflection coefficient of load (linear, not in db). If its an array it must be of shape:  $k \times n$ , where  $k$  is #frequency points in media, and  $n$  is *nports*

**nports** : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `load` :class:'~skrf.network.Network' object :

n-port load, where  $S = \text{Gamma0} * \text{eye}(\dots)$

### `skrf.media.rectangularWaveguide.RectangularWaveguide.match`

`RectangularWaveguide.match` (*nports=1*, *z0=None*, *\*\*kwargs*)

Perfect matched load ( $\Gamma_0 = 0$ ).

**Parameters** `nports` : int

number of ports

**z0** : number, or array-like

characteristic impedance. Default is None, in which case the Media's *z0* is used. This sets the resultant Network's *z0*.

**\*\*kwargs** : key word arguments

passed to `Network` initializer

**Returns** `match` : `Network` object

a n-port match

### Examples

```
>>> my_match = my_media.match(2, z0 = 50, name='Super Awesome Match')
```

**skrf.media.rectangularWaveguide.RectangularWaveguide.open**

RectangularWaveguide.**open** (*nports=1*, *\*\*kwargs*)

Open ( $\Gamma_0 = 1$ )

**Parameters** *nports* : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** *match* : `Network` object

a n-port open circuit

**See Also:**

`match` function called to create a ‘blank’ network

**skrf.media.rectangularWaveguide.RectangularWaveguide.resistor**

RectangularWaveguide.**resistor** (*R*, *\*args*, *\*\*kwargs*)

Resistor

**Parameters** *R* : number, array

Resistance, in Ohms. If this is an array, must be of same length as frequency vector.

**\*args, \*\*kwargs** : arguments, key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** *resistor* : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

**skrf.media.rectangularWaveguide.RectangularWaveguide.short**

RectangularWaveguide.**short** (*nports=1*, *\*\*kwargs*)

Short ( $\Gamma_0 = -1$ )

**Parameters** *nports* : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** *match* : `Network` object

a n-port short circuit

**See Also:**

`match` function called to create a ‘blank’ network

### skrf.media.rectangularWaveguide.RectangularWaveguide.shunt

RectangularWaveguide.**shunt** (*ntwk*, **\*\*kwargs**)

Shunts a Network

This creates a `tee()` and connects connects *ntwk* to port 1, and returns the result

**Parameters** *ntwk* : Network object

**\*\*kwargs** : keyword arguments

passed to `tee()`

**Returns** `shunted_ntwk` : Network object

a shunted a ntwk. The resultant `shunted_ntwk` will have  $(2 + \text{ntwk.number\_of\_ports} - 1)$  ports.

### skrf.media.rectangularWaveguide.RectangularWaveguide.shunt\_capacitor

RectangularWaveguide.**shunt\_capacitor** (*C*, *\*args*, **\*\*kwargs**)

Shunted capacitor

**Parameters** *C* : number, array-like

Capacitance in Farads.

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:`delay_open`

**Returns** `shunt_capacitor` : Network object

shunted capcitor(2-port)

#### Notes

This calls:

```
shunt(capacitor(C, *args, **kwargs))
```

### skrf.media.rectangularWaveguide.RectangularWaveguide.shunt\_delay\_load

RectangularWaveguide.**shunt\_delay\_load** (*\*args*, **\*\*kwargs**)

Shunted delayed load

**Parameters** **\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:`delay_load`

**Returns** `shunt_delay_load` : Network object

a shunted delayed load (2-port)

**Notes**

This calls:

```
shunt(delay_load(*args, **kwargs))
```

**skrf.media.rectangularWaveguide.RectangularWaveguide.shunt\_delay\_open**

RectangularWaveguide.**shunt\_delay\_open**(\*args, \*\*kwargs)

Shunted delayed open

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_delay\_open** : Network object

shunted delayed open (2-port)

**Notes**

This calls:

```
shunt(delay_open(*args, **kwargs))
```

**skrf.media.rectangularWaveguide.RectangularWaveguide.shunt\_delay\_short**

RectangularWaveguide.**shunt\_delay\_short**(\*args, \*\*kwargs)

Shunted delayed short

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_delay\_load** : Network object

shunted delayed open (2-port)

**Notes**

This calls:

```
shunt(delay_short(*args, **kwargs))
```

**skrf.media.rectangularWaveguide.RectangularWaveguide.shunt\_inductor**

RectangularWaveguide.**shunt\_inductor**(L, \*args, \*\*kwargs)

Shunted inductor

**Parameters** L : number, array-like

Inductance in Farads.

\*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** `shunt_inductor` : `Network` object  
shunted inductor(2-port)

#### Notes

This calls:

```
shunt(inductor(C, *args, **kwargs))
```

### `skrf.media.rectangularWaveguide.RectangularWaveguide.splitter`

`RectangularWaveguide.splitter` (*nports*, *\*\*kwargs*)  
Ideal, lossless n-way splitter.

**Parameters** `nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `tee` : `Network` object

a n-port splitter

#### See Also:

`match` called to create a ‘blank’ network

### `skrf.media.rectangularWaveguide.RectangularWaveguide.tee`

`RectangularWaveguide.tee` (*\*\*kwargs*)  
Ideal, lossless tee. (3-port splitter)

**Parameters** **\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `tee` : `Network` object

a 3-port splitter

#### See Also:

`splitter` this just calls `splitter(3)`

`match` called to create a ‘blank’ network

### `skrf.media.rectangularWaveguide.RectangularWaveguide.theta_2_d`

`RectangularWaveguide.theta_2_d` (*theta*, *deg=True*)  
Converts electrical length to physical distance.

The given electrical length is to be at the center frequency.

**Parameters** **theta** : number  
 electrical length, at band center (see `deg` for unit)

**deg** : Boolean  
 is theta in degrees?

**Returns** **d** : number  
 physical distance in meters

#### `skrf.media.rectangularWaveguide.RectangularWaveguide.thru`

`RectangularWaveguide.thru` (*\*\*kwargs*)  
 Matched transmission line of length 0.

**Parameters** *\*\*kwargs* : key word arguments  
 passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **thru** : `Network` object  
 matched transmission line of 0 length

**See Also:**

`line` this just calls `line(0)`

#### `skrf.media.rectangularWaveguide.RectangularWaveguide.white_gaussian_polar`

`RectangularWaveguide.white_gaussian_polar` (*phase\_dev, mag\_dev, n\_ports=1, \*\*kwargs*)  
 Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

**Parameters** **phase\_mag** : number  
 standard deviation of magnitude

**phase\_dev** : number  
 standard deviation of phase

**n\_ports** : int  
 number of ports.

*\*\*kwargs* : passed to `Network`  
 initializer

**Returns** **result** : `Network` object  
 a noise network

### skrf.media.rectangularWaveguide.RectangularWaveguide.write\_csv

RectangularWaveguide.**write\_csv** (*filename='f, gamma, z0.csv'*)  
 write this media's frequency, z0, and gamma to a csv file.

**Parameters** **filename** : string  
 file name to write out data to

**See Also:**

**from\_csv** class method to initialize Media object from a csv file written from this function

### skrf.media.cpw.CPW

**class** skrf.media.cpw.**CPW** (*frequency, w, s, ep\_r, t=None, rho=None, \*args, \*\*kwargs*)  
 Coplanar waveguide class

This class was made from the technical documentation <sup>35</sup> provided by the qucs project <sup>36</sup>. The variables and properties of this class are coincident with their derivations.

**Attributes**

<code>K_ratio</code>	intermediary parameter. see qucs docs on cpw lines.
<code>alpha_conductor</code>	Losses due to conductor resistivity
<code>characteristic_impedance</code>	Characterisitic impedance
<code>ep_re</code>	intermediary parameter. see qucs docs on cpw lines.
<code>k1</code>	intermediary parameter. see qucs docs on cpw lines.
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

### skrf.media.cpw.CPW.K\_ratio

**CPW.K\_ratio**  
 intermediary parameter. see qucs docs on cpw lines.

### skrf.media.cpw.CPW.alpha\_conductor

**CPW.alpha\_conductor**  
 Losses due to conductor resistivity

**Returns** **alpha\_conductor** : array-like  
 lossyness due to conductor losses

**See Also :**

\_\_\_\_\_ :  
**surface\_resistivity** : calculates surface resistivity

---

<sup>35</sup> <http://qucs.sourceforge.net/docs/technical.pdf>  
<sup>36</sup> <http://www.qucs.sourceforge.net/>



**skrf.media.cpw.CPW.characteristic\_impedance****CPW.characteristic\_impedance**

Characterisitic impedance

The `characteristic_impedance` can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `characteristic_impedance` : `numpy.ndarray`

**skrf.media.cpw.CPW.ep\_re****CPW.ep\_re**

intermediary parameter. see qucs docs on cpw lines.

**skrf.media.cpw.CPW.k1****CPW.k1**

intermediary parameter. see qucs docs on cpw lines.

**skrf.media.cpw.CPW.propagation\_constant****CPW.propagation\_constant**

Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `propagation_constant` : `numpy.ndarray`

complex propagation constant for this media

**Notes**

*propagation\_constant* must adhere to the following convention,

- positive real(propagation\_constant) = attenuation
- positive imag(propagation\_constant) = forward propagation

**skrf.media.cpw.CPW.z0****CPW.z0**

Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is `None` then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `port_impedance` : `numpy.ndarray`  
the media's port impedance

## Methods

<code>z0</code>	Characterisitic impedance
<code>__init__</code>	Coplanar Waveguide initializer
<code>capacitor</code>	Capacitor
<code>delay_load</code>	Delayed load
<code>delay_open</code>	Delayed open transmission line
<code>delay_short</code>	Delayed Short
<code>electrical_length</code>	calculates the electrical length for a given distance, at
<code>from_csv</code>	create a Media from numerical values stored in a csv file.
<code>gamma</code>	Propagation constant ..
<code>guess_length_of_delay_short</code>	Guess physical length of a delay short.
<code>impedance_mismatch</code>	Two-port network for an impedance miss-match
<code>inductor</code>	Inductor
<code>line</code>	Matched transmission line of given length
<code>load</code>	Load of given reflection coefficient.
<code>match</code>	Perfect matched load ( $\Gamma_0 = 0$ ).
<code>open</code>	Open ( $\Gamma_0 = 1$ )
<code>resistor</code>	Resistor
<code>short</code>	Short ( $\Gamma_0 = -1$ )
<code>shunt</code>	Shunts a <code>Network</code>
<code>shunt_capacitor</code>	Shunted capacitor
<code>shunt_delay_load</code>	Shunted delayed load
<code>shunt_delay_open</code>	Shunted delayed open
<code>shunt_delay_short</code>	Shunted delayed short
<code>shunt_inductor</code>	Shunted inductor
<code>splitter</code>	Ideal, lossless n-way splitter.
<code>tee</code>	Ideal, lossless tee.
<code>theta_2_d</code>	Converts electrical length to physical distance.
<code>thru</code>	Matched transmission line of length 0.
<code>white_gaussian_polar</code>	Complex zero-mean gaussian white-noise network.
<code>write_csv</code>	write this media's frequency, z0, and gamma to a csv file.

### skrf.media.cpw.CPW.Z0

`CPW.Z0()`  
Characterisitic impedance

### skrf.media.cpw.CPW.\_\_init\_\_

`CPW.__init__(frequency, w, s, ep_r, t=None, rho=None, *args, **kwargs)`  
Coplanar Waveguide initializer

**Parameters** **frequency** : `Frequency` object  
 frequency band of this transmission line medium

**w** : number, or array-like  
 width of center conductor, in m.

**s** : number, or array-like  
 width of gap, in m.

**ep\_r** : number, or array-like  
 relative permativity of substrate

**t** : number, or array-like, optional  
 conductor thickness, in m.

**rho**: number, or array-like, optional :  
 resistivity of conductor (None)

### `skrf.media.cpw.CPW.capacitor`

`CPW.capacitor` (*C*, **\*\*kwargs**)  
 Capacitor

**Parameters** **C** : number, array  
 Capacitance, in Farads. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments  
 passed to `match()`, which is called initially to create a 'blank' network.

**Returns** **capacitor** : a 2-port `Network`

**See Also:**

`match` function called to create a 'blank' network

### `skrf.media.cpw.CPW.delay_load`

`CPW.delay_load` (*Gamma0*, *d*, *unit='m'*, **\*\*kwargs**)  
 Delayed load

A load with reflection coefficient *Gamma0* at the end of a matched line of length *d*.

**Parameters** **Gamma0** : number, array-like  
 reflection coefficient of load (not in dB)

**d** : number  
 the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees

- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `delay_load` : `Network` object

a delayed load

**See Also:**

`line` creates the network for line

`load` creates the network for the load

### Notes

This calls

```
line(d,unit, **kwargs) ** load(Gamma0, **kwargs)
```

### Examples

```
>>> my_media.delay_load(-.5, 90, 'deg', z0=50)
```

### `skrf.media.cpw.CPW.delay_open`

`CPW.delay_open` (*d*, *unit='m'*, **\*\*kwargs**)

Delayed open transmission line

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `delay_open` : `Network` object

a delayed open

**See Also:**

`delay_load` `delay_short` just calls this function

**skrf.media.cpw.CPW.delay\_short****CPW.delay\_short** (*d*, *unit='m'*, *\*\*kwargs*)

Delayed Short

A transmission line of given length terminated with a short.

**Parameters** **d** : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **delay\_short** : `Network` object

a delayed short

**See Also:****delay\_load** `delay_short` just calls this function**skrf.media.cpw.CPW.electrical\_length****CPW.electrical\_length** (*d*, *deg=False*)

calculates the electrical length for a given distance, at the center frequency.

**Parameters** **d**: number or array-like :

delay distance, in meters

**deg**: Boolean :

return electral length in deg?

**Returns** **theta**: number or array-like :

electrical length in radians or degrees, depending on value of deg.

**skrf.media.cpw.CPW.from\_csv****classmethod** **CPW.from\_csv** (*filename*, *\*args*, *\*\*kwargs*)

create a Media from numerical values stored in a csv file.

the csv file format must be written by the function `write_csv()` which produces the following format

```
f[$unit], Re(Z0), Im(Z0), Re(gamma), Im(gamma), Re(port Z0), Im(port Z0) 1, 1, 1, 1, 1, 1, 1 2, 1,
1, 1, 1, 1, 1 .....
```

### skrf.media.cpw.CPW.gamma

CPW.**gamma** ()  
Propagation constant

**See Also:**

[alpha\\_conductor](#) calculates losses to conductors

### skrf.media.cpw.CPW.guess\_length\_of\_delay\_short

CPW.**guess\_length\_of\_delay\_short** (*aNtwk*)  
Guess physical length of a delay short.

Unwraps the phase and determines the slope, which is then used in conjunction with [propagation\\_constant](#) to estimate the physical distance to the short.

**Parameters** *aNtwk* : *Network* object

(note: if this is a measurement it needs to be normalized to the reference plane)

### skrf.media.cpw.CPW.impedance\_mismatch

CPW.**impedance\_mismatch** (*z1*, *z2*, *\*\*kwargs*)  
Two-port network for an impedance miss-match

**Parameters** *z1* : number, or array-like

complex impedance of port 1

*z2* : number, or array-like

complex impedance of port 2

**\*\*kwargs** : key word arguments

passed to [match\(\)](#), which is called initially to create a 'blank' network.

**Returns** *missmatch* : *Network* object

a 2-port network representing the impedance mismatch

**See Also:**

[match](#) called to create a 'blank' network

**Notes**

If *z1* and *z2* are arrays, they must be of same length as the `Media.frequency.npoints`

### skrf.media.cpw.CPW.inductor

CPW.**inductor** (*L*, *\*\*kwargs*)  
Inductor

**Parameters** *L* : number, array

Inductance, in Henrys. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns inductor** : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

### skrf.media.cpw.CPW.line

`CPW.line` (*d*, *unit='m'*, **\*\*kwargs**)

Matched transmission line of given length

The units of *length* are interpreted according to the value of *unit*.

**Parameters d** : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns line** : `Network` object

matched tranmission line of given length

### Examples

```
>>> my_media.line(90, 'deg', z0=50)
```

### skrf.media.cpw.CPW.load

`CPW.load` (*Gamma0*, *nports=1*, **\*\*kwargs**)

Load of given reflection coefficient.

**Parameters Gamma0** : number, array-like

Reflection coefficient of load (linear, not in db). If its an array it must be of shape:  $k \times n$ , where  $k$  is #frequency points in media, and  $n$  is *nports*

**nports** : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `load` :class:`~skrf.network.Network` object :

n-port load, where  $S = \Gamma_0 * \text{eye}(\dots)$

### skrf.media.cpw.CPW.match

CPW.**match** (*nports=1, z0=None, \*\*kwargs*)

Perfect matched load ( $\Gamma_0 = 0$ ).

**Parameters** `nports` : int

number of ports

`z0` : number, or array-like

characteristic impedance. Default is None, in which case the Media's `z0` is used. This sets the resultant Network's `z0`.

**\*\*kwargs** : key word arguments

passed to `Network` initializer

**Returns** `match` : `Network` object

a n-port match

### Examples

```
>>> my_match = my_media.match(2, z0 = 50, name='Super Awesome Match')
```

### skrf.media.cpw.CPW.open

CPW.**open** (*nports=1, \*\*kwargs*)

Open ( $\Gamma_0 = 1$ )

**Parameters** `nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `match` : `Network` object

a n-port open circuit

### See Also:

`match` function called to create a 'blank' network

### skrf.media.cpw.CPW.resistor

CPW.**resistor** (*R, \*args, \*\*kwargs*)

Resistor

**Parameters** `R` : number, array

Resistance, in Ohms. If this is an array, must be of same length as frequency vector.



**\*args, \*\*kwargs** : arguments, key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `resistor` : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

### skrf.media.cpw.CPW.short

`CPW.short` (*nports=1, \*\*kwargs*)

Short ( $\Gamma_0 = -1$ )

**Parameters** `nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `match` : `Network` object

a n-port short circuit

**See Also:**

`match` function called to create a ‘blank’ network

### skrf.media.cpw.CPW.shunt

`CPW.shunt` (*ntwk, \*\*kwargs*)

Shunts a `Network`

This creates a `tee()` and connects connects *ntwk* to port 1, and returns the result

**Parameters** `ntwk` : `Network` object

**\*\*kwargs** : keyword arguments

passed to `tee()`

**Returns** `shunted_ntwk` : `Network` object

a shunted a ntwk. The resultant shunted\_ntwk will have (2 + ntwk.number\_of\_ports -1) ports.

### skrf.media.cpw.CPW.shunt\_capacitor

`CPW.shunt_capacitor` (*C, \*args, \*\*kwargs*)

Shunted capacitor

**Parameters** `C` : number, array-like

Capacitance in Farads.

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** `shunt_capacitor` : `Network` object  
shunted capacitor(2-port)

#### Notes

This calls:

```
shunt(capacitor(C, *args, **kwargs))
```

#### `skrf.media.cpw.CPW.shunt_delay_load`

`CPW.shunt_delay_load` (*\*args*, *\*\*kwargs*)  
Shunted delayed load

**Parameters** *\*args, \*\*kwargs* : arguments, keyword arguments  
passed to func:*delay\_load*

**Returns** `shunt_delay_load` : `Network` object  
a shunted delayed load (2-port)

#### Notes

This calls:

```
shunt(delay_load(*args, **kwargs))
```

#### `skrf.media.cpw.CPW.shunt_delay_open`

`CPW.shunt_delay_open` (*\*args*, *\*\*kwargs*)  
Shunted delayed open

**Parameters** *\*args, \*\*kwargs* : arguments, keyword arguments  
passed to func:*delay\_open*

**Returns** `shunt_delay_open` : `Network` object  
shunted delayed open (2-port)

#### Notes

This calls:

```
shunt(delay_open(*args, **kwargs))
```

#### `skrf.media.cpw.CPW.shunt_delay_short`

`CPW.shunt_delay_short` (*\*args*, *\*\*kwargs*)  
Shunted delayed short

**Parameters** *\*args, \*\*kwargs* : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** `shunt_delay_load` : `Network` object  
shunted delayed open (2-port)

#### Notes

This calls:

```
shunt(delay_short(*args, **kwargs))
```

#### `skrf.media.cpw.CPW.shunt_inductor`

`CPW.shunt_inductor` (*L*, *\*args*, *\*\*kwargs*)  
Shunted inductor

**Parameters** `L` : number, array-like  
Inductance in Farads.

**\*args, \*\*kwargs** : arguments, keyword arguments  
passed to func:*delay\_open*

**Returns** `shunt_inductor` : `Network` object  
shunted inductor(2-port)

#### Notes

This calls:

```
shunt(inductor(C, *args, **kwargs))
```

#### `skrf.media.cpw.CPW.splitter`

`CPW.splitter` (*nports*, *\*\*kwargs*)  
Ideal, lossless n-way splitter.

**Parameters** `nports` : int  
number of ports

**\*\*kwargs** : key word arguments  
passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `tee` : `Network` object  
a n-port splitter

**See Also:**

`match` called to create a ‘blank’ network

### skrf.media.cpw.CPW.tee

CPW.**tee** (*\*\*kwargs*)

Ideal, lossless tee. (3-port splitter)

**Parameters** **\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **tee** : `Network` object

a 3-port splitter

**See Also:**

`splitter` this just calls `splitter(3)`

`match` called to create a ‘blank’ network

### skrf.media.cpw.CPW.theta\_2\_d

CPW.**theta\_2\_d** (*theta, deg=True*)

Converts electrical length to physical distance.

The given electrical length is to be at the center frequency.

**Parameters** **theta** : number

electrical length, at band center (see `deg` for unit)

**deg** : Boolean

is theta in degrees?

**Returns** **d** : number

physical distance in meters

### skrf.media.cpw.CPW.thru

CPW.**thru** (*\*\*kwargs*)

Matched transmission line of length 0.

**Parameters** **\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **thru** : `Network` object

matched transmission line of 0 length

**See Also:**

`line` this just calls `line(0)`

**skrf.media.cpw.CPW.white\_gaussian\_polar**

CPW.**white\_gaussian\_polar** (*phase\_dev, mag\_dev, n\_ports=1, \*\*kwargs*)

Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

**Parameters** **phase\_mag** : number  
 standard deviation of magnitude

**phase\_dev** : number  
 standard deviation of phase

**n\_ports** : int  
 number of ports.

**\*\*kwargs** : passed to [Network](#)  
 initializer

**Returns** **result** : [Network](#) object  
 a noise network

**skrf.media.cpw.CPW.write\_csv**

CPW.**write\_csv** (*filename='f, gamma, z0.csv'*)

write this media’s frequency, z0, and gamma to a csv file.

**Parameters** **filename** : string  
 file name to write out data to

**See Also:**

[from\\_csv](#) class method to initialize Media object from a csv file written from this function

**skrf.media.freespace.Freespace**

**class** [skrf.media.freespace.Freespace](#) (*frequency, ep\_r=1, mu\_r=1, \*args, \*\*kwargs*)

Represents a plane-wave in a homogeneous freespace, defined by the space’s relative permativity and relative permeability.

The field properties of space are related to a distributed circuit transmission line model given in circuit theory by:

Circuit Property	Field Property
distributed_capacitance	$\text{real}(ep\_0 * ep\_r)$
distributed_resistance	$\text{imag}(ep\_0 * ep\_r)$
distributed_inductance	$\text{real}(mu\_0 * mu\_r)$
distributed_conductance	$\text{imag}(mu\_0 * mu\_r)$

**This class’s inheritance is;** [Media](#)->[DistributedCircuit](#)->[Freespace](#)

## Attributes

<code>Y</code>	Distributed Admittance, $Y'$
<code>Z</code>	Distributed Impedance, $Z'$
<code>characteristic_impedance</code>	Characterisitic impedance
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

### `skrf.media.freespace.Freespace.Y`

`Freespace.Y`

Distributed Admittance,  $Y'$

Defined as

$$Y' = G' + j\omega C'$$

**Returns** `Y` : `numpy.ndarray`

Distributed Admittance in units of S/m

### `skrf.media.freespace.Freespace.Z`

`Freespace.Z`

Distributed Impedance,  $Z'$

Defined as

$$Z' = R' + j\omega I'$$

**Returns** `Z` : `numpy.ndarray`

Distributed impedance in units of ohm/m

### `skrf.media.freespace.Freespace.characteristic_impedance`

`Freespace.characteristic_impedance`

Characterisitic impedance

The `characteristic_impedance` can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `characteristic_impedance` : `numpy.ndarray`

### `skrf.media.freespace.Freespace.propagation_constant`

`Freespace.propagation_constant`

Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `propagation_constant` : `numpy.ndarray`  
 complex propagation constant for this media

## Notes

*propagation\_constant* must adhere to the following convention,

- positive `real(propagation_constant)` = attenuation
- positive `imag(propagation_constant)` = forward propagation

## `skrf.media.freespace.Freespace.z0`

### `Freespace.z0`

Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is `None` then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

**Returns** `port_impedance` : `numpy.ndarray`  
 the media's port impedance

## Methods

<code>z0</code>	Characteristic Impedance, $Z_0$
<code>__init__</code>	Freespace initializer
<code>capacitor</code>	Capacitor
<code>delay_load</code>	Delayed load
<code>delay_open</code>	Delayed open transmission line
<code>delay_short</code>	Delayed Short
<code>electrical_length</code>	calculates the electrical length for a given distance, at
<code>from_Media</code>	Initializes a <code>DistributedCircuit</code> from an existing
<code>from_csv</code>	create a <code>Media</code> from numerical values stored in a csv file.
<code>gamma</code>	Propagation Constant, $\gamma$
<code>guess_length_of_delay_short</code>	Guess physical length of a delay short.
<code>impedance_mismatch</code>	Two-port network for an impedance miss-match
<code>inductor</code>	Inductor
<code>line</code>	Matched transmission line of given length
<code>load</code>	Load of given reflection coefficient.
<code>match</code>	Perfect matched load ( $\Gamma_0 = 0$ ).
Continued on next page	

**Table 3.53 – continued from previous page**

<code>open</code>	Open ( $\Gamma_0 = 1$ )
<code>resistor</code>	Resistor
<code>short</code>	Short ( $\Gamma_0 = -1$ )
<code>shunt</code>	Shunts a <code>Network</code>
<code>shunt_capacitor</code>	Shunted capacitor
<code>shunt_delay_load</code>	Shunted delayed load
<code>shunt_delay_open</code>	Shunted delayed open
<code>shunt_delay_short</code>	Shunted delayed short
<code>shunt_inductor</code>	Shunted inductor
<code>splitter</code>	Ideal, lossless n-way splitter.
<code>tee</code>	Ideal, lossless tee.
<code>theta_2_d</code>	Converts electrical length to physical distance.
<code>thru</code>	Matched transmission line of length 0.
<code>white_gaussian_polar</code>	Complex zero-mean gaussian white-noise network.
<code>write_csv</code>	write this media's frequency, <code>z0</code> , and <code>gamma</code> to a csv file.

**skrf.media.freespace.Freespace.Z0**

`Freespace.Z0()`  
 Characteristic Impedance, `Z0`

$$Z_0 = \sqrt{\frac{Z'}{Y'}}$$

**Returns** `Z0` : `numpy.ndarray`  
 Characteristic Impedance in units of ohms

**skrf.media.freespace.Freespace.\_\_init\_\_**

`Freespace.__init__(frequency, ep_r=1, mu_r=1, *args, **kwargs)`  
 Freespace initializer

**Parameters** `frequency` : `Frequency` object  
 frequency band of this transmission line medium  
`ep_r` : number, array-like  
 complex relative permativity  
`mu_r` : number, array-like  
 possibly complex, relative permiability  
`*args, **kwargs` : arguments and keyword arguments

**Notes**

The distributed circuit parameters are related to a space's field properties by



Circuit Property	Field Property
distributed_capacitance	real( $\epsilon_{p_0} * \epsilon_{p_r}$ )
distributed_resistance	imag( $\epsilon_{p_0} * \epsilon_{p_r}$ )
distributed_inductance	real( $\mu_{0_0} * \mu_{0_r}$ )
distributed_conductance	imag( $\mu_{0_0} * \mu_{0_r}$ )

### skrf.media.freespace.Freespace.capacitor

Freespace.**capacitor** (*C*, **\*\*kwargs**)

Capacitor

**Parameters** *C* : number, array

Capacitance, in Farads. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **capacitor** : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

### skrf.media.freespace.Freespace.delay\_load

Freespace.**delay\_load** (*Gamma0*, *d*, *unit='m'*, **\*\*kwargs**)

Delayed load

A load with reflection coefficient *Gamma0* at the end of a matched line of length *d*.

**Parameters** **Gamma0** : number, array-like

reflection coefficient of load (not in dB)

**d** : number

the length of transmissin line (see unit argument)

**unit** : ['m', 'deg', 'rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** **delay\_load** : `Network` object

a delayed load

**See Also:**

`line` creates the network for line

`load` creates the network for the load

## Notes

This calls

```
line(d,unit, **kwargs) ** load(Gamma0, **kwargs)
```

## Examples

```
>>> my_media.delay_load(-.5, 90, 'deg', z0=50)
```

## skrf.media.freespace.Freespace.delay\_open

`Freespace.delay_open` (*d*, *unit*='m', *\*\*kwargs*)

Delayed open transmission line

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `delay_open` : `Network` object

a delayed open

**See Also:**

`delay_load` `delay_short` just calls this function

## skrf.media.freespace.Freespace.delay\_short

`Freespace.delay_short` (*d*, *unit*='m', *\*\*kwargs*)

Delayed Short

A transmission line of given length terminated with a short.

**Parameters** *d* : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `delay_short` : `Network` object

a delayed short

**See Also:**

`delay_load` `delay_short` just calls this function

### `skrf.media.freespace.Freespace.electrical_length`

`Freespace.electrical_length` (*d*, *deg=False*)

calculates the electrical length for a given distance, at the center frequency.

**Parameters** **d**: number or array-like :

delay distance, in meters

**deg**: Boolean :

return electral length in deg?

**Returns** **theta**: number or array-like :

electrical length in radians or degrees, depending on value of deg.

### `skrf.media.freespace.Freespace.from_Media`

**classmethod** `Freespace.from_Media` (*my\_media*, *\*args*, *\*\*kwargs*)

Initializes a `DistributedCircuit` from an existing `:class:'~skrf.media.media.Media'` instance.

### `skrf.media.freespace.Freespace.from_csv`

**classmethod** `Freespace.from_csv` (*filename*, *\*args*, *\*\*kwargs*)

create a `Media` from numerical values stored in a csv file.

the csv file format must be written by the function `write_csv()` which produces the following format

```
f[$unit], Re(Z0), Im(Z0), Re(gamma), Im(gamma), Re(port Z0), Im(port Z0) 1, 1, 1, 1, 1, 1, 1 2, 1,
1, 1, 1, 1, 1 .....
```

### `skrf.media.freespace.Freespace.gamma`

`Freespace.gamma` ()

Propagation Constant,  $\gamma$

Defined as,

$$\gamma = \sqrt{Z'Y'}$$

**Returns** **gamma** : `numpy.ndarray`

Propagation Constant,

## Notes

The components of propagation constant are interpreted as follows:  
positive real( $\gamma$ ) = attenuation positive imag( $\gamma$ ) = forward propagation

### `skrf.media.freespace.Freespace.guess_length_of_delay_short`

`Freespace.guess_length_of_delay_short` (*aNtwk*)  
Guess physical length of a delay short.

Unwraps the phase and determines the slope, which is then used in conjunction with `propagation_constant` to estimate the physical distance to the short.

**Parameters** `aNtwk` : `Network` object

(note: if this is a measurement it needs to be normalized to the reference plane)

### `skrf.media.freespace.Freespace.impedance_mismatch`

`Freespace.impedance_mismatch` (*z1*, *z2*, *\*\*kwargs*)  
Two-port network for an impedance miss-match

**Parameters** `z1` : number, or array-like

complex impedance of port 1

`z2` : number, or array-like

complex impedance of port 2

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `missmatch` : `Network` object

a 2-port network representing the impedance mismatch

## See Also:

`match` called to create a 'blank' network

## Notes

If `z1` and `z2` are arrays, they must be of same length as the `Media.frequency.npoints`

### `skrf.media.freespace.Freespace.inductor`

`Freespace.inductor` (*L*, *\*\*kwargs*)  
Inductor

**Parameters** `L` : number, array

Inductance, in Henrys. If this is an array, must be of same length as frequency vector.

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `inductor` : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

### `skrf.media.freespace.Freespace.line`

`Freespace.line` (*d*, *unit='m'*, *\*\*kwargs*)

Matched transmission line of given length

The units of *length* are interpreted according to the value of *unit*.

**Parameters** `d` : number

the length of transmissin line (see unit argument)

**unit** : ['m','deg','rad']

**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `line` : `Network` object

matched tranmission line of given length

### Examples

```
>>> my_media.line(90, 'deg', z0=50)
```

### `skrf.media.freespace.Freespace.load`

`Freespace.load` (*Gamma0*, *nports=1*, *\*\*kwargs*)

Load of given reflection coefficient.

**Parameters** `Gamma0` : number, array-like

Reflection coefficient of load (linear, not in db). If its an array it must be of shape:  $k \times n$ , where  $k$  is #frequency points in media, and  $n$  is *nports*

**nports** : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `load` :class:‘~skrf.network.Network’ object :

n-port load, where  $S = \text{Gamma0} * \text{eye}(\dots)$

**skrf.media.freespace.Freespace.match**

`Freespace.match` (*nports=1, z0=None, \*\*kwargs*)

Perfect matched load ( $\Gamma_0 = 0$ ).

**Parameters** `nports` : int

number of ports

`z0` : number, or array-like

characteristic impedance. Default is None, in which case the Media's `z0` is used. This sets the resultant Network's `z0`.

**\*\*kwargs** : key word arguments

passed to `Network` initializer

**Returns** `match` : `Network` object

a n-port match

**Examples**

```
>>> my_match = my_media.match(2, z0 = 50, name='Super Awesome Match')
```

**skrf.media.freespace.Freespace.open**

`Freespace.open` (*nports=1, \*\*kwargs*)

Open ( $\Gamma_0 = 1$ )

**Parameters** `nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `match` : `Network` object

a n-port open circuit

**See Also:**

`match` function called to create a 'blank' network

**skrf.media.freespace.Freespace.resistor**

`Freespace.resistor` (*R, \*args, \*\*kwargs*)

Resistor

**Parameters** `R` : number, array

Resistance, in Ohms. If this is an array, must be of same length as frequency vector.

**\*args, \*\*kwargs** : arguments, key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

**Returns** `resistor` : a 2-port `Network`

**See Also:**

`match` function called to create a ‘blank’ network

### `skrf.media.freespace.Freespace.short`

`Freespace.short` (*nports=1, \*\*kwargs*)

Short ( $\Gamma_0 = -1$ )

**Parameters** `nports` : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `match` : `Network` object

a n-port short circuit

**See Also:**

`match` function called to create a ‘blank’ network

### `skrf.media.freespace.Freespace.shunt`

`Freespace.shunt` (*ntwk, \*\*kwargs*)

Shunts a `Network`

This creates a `tee()` and connects connects *ntwk* to port 1, and returns the result

**Parameters** `ntwk` : `Network` object

**\*\*kwargs** : keyword arguments

passed to `tee()`

**Returns** `shunted_ntwk` : `Network` object

a shunted a `ntwk`. The resultant `shunted_ntwk` will have  $(2 + \text{ntwk.number\_of\_ports} - 1)$  ports.

### `skrf.media.freespace.Freespace.shunt_capacitor`

`Freespace.shunt_capacitor` (*C, \*args, \*\*kwargs*)

Shunted capacitor

**Parameters** `C` : number, array-like

Capacitance in Farads.

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** `shunt_capacitor` : `Network` object

shunted capacitor(2-port)

### Notes

This calls:

```
shunt(capacitor(C,*args, **kwargs))
```

### skrf.media.freespace.Freespace.shunt\_delay\_load

Freespace.**shunt\_delay\_load**(\*args, \*\*kwargs)

Shunted delayed load

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_load*

**Returns** **shunt\_delay\_load** : *Network* object

a shunted delayed load (2-port)

### Notes

This calls:

```
shunt(delay_load(*args, **kwargs))
```

### skrf.media.freespace.Freespace.shunt\_delay\_open

Freespace.**shunt\_delay\_open**(\*args, \*\*kwargs)

Shunted delayed open

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_delay\_open** : *Network* object

shunted delayed open (2-port)

### Notes

This calls:

```
shunt(delay_open(*args, **kwargs))
```

### skrf.media.freespace.Freespace.shunt\_delay\_short

Freespace.**shunt\_delay\_short**(\*args, \*\*kwargs)

Shunted delayed short

**Parameters** \*args,\*\*kwargs : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** **shunt\_delay\_load** : *Network* object

shunted delayed open (2-port)



**Notes**

This calls:

```
shunt(delay_short(*args, **kwargs))
```

**skrf.media.freespace.Freespace.shunt\_inductor**

`Freespace.shunt_inductor` (*L*, \*args, \*\*kwargs)

Shunted inductor

**Parameters** *L* : number, array-like

Inductance in Farads.

**\*args, \*\*kwargs** : arguments, keyword arguments

passed to func:*delay\_open*

**Returns** `shunt_inductor` : `Network` object

shunted inductor(2-port)

**Notes**

This calls:

```
shunt(inductor(C, *args, **kwargs))
```

**skrf.media.freespace.Freespace.splitter**

`Freespace.splitter` (*nports*, \*\*kwargs)

Ideal, lossless n-way splitter.

**Parameters** *nports* : int

number of ports

**\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `tee` : `Network` object

a n-port splitter

**See Also:**

`match` called to create a ‘blank’ network

**skrf.media.freespace.Freespace.tee**

`Freespace.tee` (\*\*kwargs)

Ideal, lossless tee. (3-port splitter)

**Parameters** **\*\*kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `tee` : `Network` object  
a 3-port splitter

**See Also:**

`splitter` this just calls `splitter(3)`  
`match` called to create a ‘blank’ network

#### `skrf.media.freespace.Freespace.theta_2_d`

`Freespace.theta_2_d` (*theta*, *deg=True*)  
Converts electrical length to physical distance.

The given electrical length is to be at the center frequency.

**Parameters** `theta` : number  
electrical length, at band center (see `deg` for unit)  
`deg` : Boolean  
is theta in degrees?

**Returns** `d` : number  
physical distance in meters

#### `skrf.media.freespace.Freespace.thru`

`Freespace.thru` (*\*\*kwargs*)  
Matched transmission line of length 0.

**Parameters** *\*\*kwargs* : key word arguments  
passed to `match()`, which is called initially to create a ‘blank’ network.

**Returns** `thru` : `Network` object  
matched transmission line of 0 length

**See Also:**

`line` this just calls `line(0)`

#### `skrf.media.freespace.Freespace.white_gaussian_polar`

`Freespace.white_gaussian_polar` (*phase\_dev*, *mag\_dev*, *n\_ports=1*, *\*\*kwargs*)  
Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

**Parameters** `phase_mag` : number  
standard deviation of magnitude  
`phase_dev` : number  
standard deviation of phase

**n\_ports** : int

number of ports.

**\*\*kwargs** : passed to `Network`

initializer

**Returns** **result** : `Network` object

a noise network

#### `skrf.media.freespace.Freespace.write_csv`

`Freespace.write_csv` (*filename*='f, gamma, z0.csv')

write this media's frequency, *z0*, and gamma to a csv file.

**Parameters** **filename** : string

file name to write out data to

**See Also:**

`from_csv` class method to initialize `Media` object from a csv file written from this function

## 3.12 Indices and tables

- *genindex*
- *modindex*
- *search*



# PYTHON MODULE INDEX

## S

- skrf.calibration, 166
- skrf.calibration.calibration, 166
- skrf.calibration.calibrationAlgorithms,  
173
- skrf.calibration.calibrationFunctions,  
177
- skrf.constants, 155
- skrf.frequency, 53
- skrf.io, 158
- skrf.io.csv, 165
- skrf.io.general, 158
- skrf.io.touchstone, 162
- skrf.mathFunctions, 144
- skrf.media, 185
- skrf.network, 57
- skrf.networkSet, 132
- skrf.plotting, 138
- skrf.tlineFunctions, 147
- skrf.util, 156



# INDEX

## Symbols

`__init__()` (skrf.calibration.calibration.Calibration method), 169, 181  
`__init__()` (skrf.frequency.Frequency method), 56  
`__init__()` (skrf.io.touchstone.Touchstone method), 163  
`__init__()` (skrf.media.cpw.CPW method), 230  
`__init__()` (skrf.media.distributedCircuit.DistributedCircuit method), 202  
`__init__()` (skrf.media.freespace.Freespace method), 244  
`__init__()` (skrf.media.media.Media method), 188  
`__init__()` (skrf.media.rectangularWaveguide.RectangularWaveguide method), 216  
`__init__()` (skrf.network.Network method), 72  
`__init__()` (skrf.networkSet.NetworkSet method), 134

## A

`a` (skrf.network.Network attribute), 60  
`a_arcl` (skrf.network.Network attribute), 60  
`a_arcl_unwrap` (skrf.network.Network attribute), 61  
`a_db` (skrf.network.Network attribute), 61  
`a_deg` (skrf.network.Network attribute), 61  
`a_deg_unwrap` (skrf.network.Network attribute), 61  
`a_im` (skrf.network.Network attribute), 61  
`a_mag` (skrf.network.Network attribute), 61  
`a_rad` (skrf.network.Network attribute), 62  
`a_rad_unwrap` (skrf.network.Network attribute), 62  
`a_re` (skrf.network.Network attribute), 62  
`abc_2_coefs_dict()` (in module skrf.calibration.calibrationAlgorithms), 177  
`add_markers_to_lines()` (in module skrf.plotting), 143  
`add_noise_polar()` (skrf.network.Network method), 73, 123  
`add_noise_polar_flatband()` (skrf.network.Network method), 73, 123  
`alpha_conductor` (skrf.media.cpw.CPW attribute), 228  
`apply_cal()` (skrf.calibration.calibration.Calibration method), 169, 181  
`apply_cal_to_all_in_dir()` (skrf.calibration.calibration.Calibration method), 170, 181  
`average()` (in module skrf.network), 131

## B

`biased_error()` (skrf.calibration.calibration.Calibration method), 170, 182

## C

`caled_ntwk_sets` (skrf.calibration.calibration.Calibration attribute), 167, 179  
`caled_ntwks` (skrf.calibration.calibration.Calibration attribute), 167, 179  
`Calibration` (class in skrf.calibration.calibration), 166, 178  
`calibration_algorithm_dict` (skrf.calibration.calibration.Calibration attribute), 167, 179  
`capacitor()` (skrf.media.cpw.CPW method), 231  
`capacitor()` (skrf.media.distributedCircuit.DistributedCircuit method), 203  
`capacitor()` (skrf.media.freespace.Freespace method), 245  
`capacitor()` (skrf.media.media.Media method), 189  
`capacitor()` (skrf.media.rectangularWaveguide.RectangularWaveguide method), 217  
`cartesian_product_calibration_set()` (in module skrf.calibration.calibrationFunctions), 177  
`cascade()` (in module skrf.network), 119  
`center` (skrf.frequency.Frequency attribute), 54  
`characteristic_impedance` (skrf.media.cpw.CPW attribute), 229  
`characteristic_impedance` (skrf.media.distributedCircuit.DistributedCircuit attribute), 200  
`characteristic_impedance` (skrf.media.freespace.Freespace attribute), 242  
`characteristic_impedance` (skrf.media.media.Media attribute), 186  
`characteristic_impedance` (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 214  
`coefs` (skrf.calibration.calibration.Calibration attribute), 167, 179  
`complex2Scalar()` (in module skrf.mathFunctions), 146  
`complex_2_db()` (in module skrf.mathFunctions), 144

complex\_2\_degree() (in module skrf.mathFunctions), 145  
 complex\_2\_magnitude() (in module skrf.mathFunctions), 144, 145  
 complex\_2\_radian() (in module skrf.mathFunctions), 144  
 complex\_2\_reim() (in module skrf.mathFunctions), 144  
 connect() (in module skrf.network), 117  
 connect\_s() (in module skrf.network), 124  
 copy() (skrf.frequency.Frequency method), 57  
 copy() (skrf.network.Network method), 73  
 copy() (skrf.networkSet.NetworkSet method), 134  
 copy\_from() (skrf.network.Network method), 74  
 CPW (class in skrf.media.cpw), 228

## D

db\_2\_np() (in module skrf.mathFunctions), 146  
 de\_embed() (in module skrf.network), 119  
 degree\_2\_radian() (in module skrf.mathFunctions), 145  
 delay\_load() (skrf.media.cpw.CPW method), 231  
 delay\_load() (skrf.media.distributedCircuit.DistributedCircuit method), 203  
 delay\_load() (skrf.media.freespace.Freespace method), 245  
 delay\_load() (skrf.media.media.Media method), 189  
 delay\_load() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 218  
 delay\_open() (skrf.media.cpw.CPW method), 232  
 delay\_open() (skrf.media.distributedCircuit.DistributedCircuit method), 204  
 delay\_open() (skrf.media.freespace.Freespace method), 246  
 delay\_open() (skrf.media.media.Media method), 190  
 delay\_open() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 218  
 delay\_short() (skrf.media.cpw.CPW method), 233  
 delay\_short() (skrf.media.distributedCircuit.DistributedCircuit method), 204  
 delay\_short() (skrf.media.freespace.Freespace method), 246  
 delay\_short() (skrf.media.media.Media method), 190  
 delay\_short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 219  
 dirac\_delta() (in module skrf.mathFunctions), 146  
 distance\_2\_electrical\_length() (in module skrf.tlineFunctions), 150  
 distributed\_circuit\_2\_propagation\_impedance() (in module skrf.tlineFunctions), 153  
 DistributedCircuit (class in skrf.media.distributedCircuit), 199

## E

eight\_term\_2\_one\_port\_coefs() (in module skrf.calibration.calibrationAlgorithms), 177  
 electrical\_length() (skrf.media.cpw.CPW method), 233

electrical\_length() (skrf.media.distributedCircuit.DistributedCircuit method), 205  
 electrical\_length() (skrf.media.freespace.Freespace method), 247  
 electrical\_length() (skrf.media.media.Media method), 191  
 electrical\_length() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 219  
 electrical\_length\_2\_distance() (in module skrf.tlineFunctions), 151  
 element\_wise\_method() (skrf.networkSet.NetworkSet method), 135  
 ep (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 214  
 ep\_re (skrf.media.cpw.CPW attribute), 229  
 error\_ntwk (skrf.calibration.calibration.Calibration attribute), 167, 179

## F

f (skrf.frequency.Frequency attribute), 54  
 f (skrf.network.Network attribute), 62  
 f\_scaled (skrf.frequency.Frequency attribute), 54  
 find\_nearest() (in module skrf.util), 157  
 find\_nearest\_index() (in module skrf.util), 157  
 flip() (in module skrf.network), 119  
 flip() (skrf.network.Network method), 74  
 freespace (class in skrf.media.freespace), 241  
 Frequency (class in skrf.frequency), 53  
 frequency (skrf.network.Network attribute), 62  
 from\_csv() (skrf.media.cpw.CPW class method), 233  
 from\_csv() (skrf.media.distributedCircuit.DistributedCircuit class method), 205  
 from\_csv() (skrf.media.freespace.Freespace class method), 247  
 from\_csv() (skrf.media.media.Media class method), 191  
 from\_csv() (skrf.media.rectangularWaveguide.RectangularWaveguide class method), 220  
 from\_f() (skrf.frequency.Frequency class method), 57  
 from\_Media() (skrf.media.distributedCircuit.DistributedCircuit class method), 205  
 from\_Media() (skrf.media.freespace.Freespace class method), 247  
 from\_zip() (skrf.networkSet.NetworkSet class method), 135  
 func\_on\_all\_figs() (in module skrf.plotting), 143  
 func\_per\_standard() (skrf.calibration.calibration.Calibration method), 170, 182

## G

gamma() (skrf.media.cpw.CPW method), 234  
 gamma() (skrf.media.distributedCircuit.DistributedCircuit method), 205  
 gamma() (skrf.media.freespace.Freespace method), 247



- Gamma0\_2\_Gamma\_in() (in module skrf.tlineFunctions), 149
- Gamma0\_2\_zin() (in module skrf.tlineFunctions), 150
- Gamma0\_2\_zl() (in module skrf.tlineFunctions), 149
- get\_comments() (skrf.io.touchstone.Touchstone method), 163
- get\_extn() (in module skrf.util), 158
- get\_fid() (in module skrf.util), 158
- get\_format() (skrf.io.touchstone.Touchstone method), 163
- get\_noise\_data() (skrf.io.touchstone.Touchstone method), 163
- get\_noise\_names() (skrf.io.touchstone.Touchstone method), 163
- get\_sparameter\_arrays() (skrf.io.touchstone.Touchstone method), 164
- get\_sparameter\_data() (skrf.io.touchstone.Touchstone method), 164
- get\_sparameter\_names() (skrf.io.touchstone.Touchstone method), 164
- guess\_length\_of\_delay\_short() (skrf.media.cpw.CPW method), 234
- guess\_length\_of\_delay\_short() (skrf.media.distributedCircuit.DistributedCircuit method), 206
- guess\_length\_of\_delay\_short() (skrf.media.freespace.Freespace method), 248
- guess\_length\_of\_delay\_short() (skrf.media.media.Media method), 191
- guess\_length\_of\_delay\_short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 220
- H**
- hfss\_touchstone\_2\_gamma\_z0() (in module skrf.io.touchstone), 164
- hfss\_touchstone\_2\_media() (in module skrf.io.touchstone), 165
- I**
- impedance\_mismatch() (skrf.media.cpw.CPW method), 234
- impedance\_mismatch() (skrf.media.distributedCircuit.DistributedCircuit method), 206
- impedance\_mismatch() (skrf.media.freespace.Freespace method), 248
- impedance\_mismatch() (skrf.media.media.Media method), 191
- impedance\_mismatch() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 220
- inductor() (skrf.media.cpw.CPW method), 234
- inductor() (skrf.media.distributedCircuit.DistributedCircuit method), 206
- inductor() (skrf.media.freespace.Freespace method), 248
- inductor() (skrf.media.media.Media method), 192
- inductor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 221
- innerconnect() (in module skrf.network), 118
- innerconnect\_s() (in module skrf.network), 125
- input\_impedance\_at\_theta() (in module skrf.tlineFunctions), 152
- interpolate() (skrf.network.Network method), 74, 120
- interpolate\_from\_f() (skrf.network.Network method), 75, 121
- interpolate\_self() (skrf.network.Network method), 75, 121
- interpolate\_self\_npoints() (skrf.network.Network method), 76
- inv (skrf.network.Network attribute), 63
- inv (skrf.networkSet.NetworkSet attribute), 133
- inv() (in module skrf.network), 124
- K**
- k0 (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 214
- k1 (skrf.media.cpw.CPW attribute), 229
- K\_ratio (skrf.media.cpw.CPW attribute), 228
- kc (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 214
- kx (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 214
- ky (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 215
- kz() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 221
- L**
- labelXAxis() (skrf.frequency.Frequency method), 57
- legend\_off() (in module skrf.plotting), 143
- line() (skrf.media.cpw.CPW method), 235
- line() (skrf.media.distributedCircuit.DistributedCircuit method), 207
- line() (skrf.media.freespace.Freespace method), 249
- line() (skrf.media.media.Media method), 192
- line() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 221
- load() (skrf.media.cpw.CPW method), 235
- load() (skrf.media.distributedCircuit.DistributedCircuit method), 207
- load() (skrf.media.freespace.Freespace method), 249
- load() (skrf.media.media.Media method), 193
- load() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 222
- load\_file() (skrf.io.touchstone.Touchstone method), 164
- load\_impedance\_2\_reflection\_coefficient() (in module skrf.tlineFunctions), 153

load\_impedance\_2\_reflection\_coefficient\_at\_theta() (in module skrf.tlineFunctions), 153

## M

match() (skrf.media.cpw.CPW method), 236  
 match() (skrf.media.distributedCircuit.DistributedCircuit method), 208  
 match() (skrf.media.freespace.Freespace method), 250  
 match() (skrf.media.media.Media method), 193  
 match() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 222  
 mean\_residuals() (skrf.calibration.calibration.Calibration method), 170, 182  
 mean\_s\_db (skrf.networkSet.NetworkSet attribute), 133  
 Media (class in skrf.media.media), 186  
 mu (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 215  
 multiplier (skrf.frequency.Frequency attribute), 54  
 multiplier\_dict (skrf.frequency.Frequency attribute), 55  
 multiply\_noise() (skrf.network.Network method), 76, 123

## N

Network (class in skrf.network), 58  
 NetworkSet (class in skrf.networkSet), 132  
 neuman() (in module skrf.mathFunctions), 146  
 now\_string() (in module skrf.util), 157  
 np\_2\_db() (in module skrf.mathFunctions), 146  
 npoints (skrf.frequency.Frequency attribute), 55  
 nports (skrf.calibration.calibration.Calibration attribute), 168, 179  
 nports (skrf.network.Network attribute), 63  
 nstandards (skrf.calibration.calibration.Calibration attribute), 168, 179  
 nudge() (skrf.network.Network method), 77, 132  
 null() (in module skrf.mathFunctions), 147  
 number\_of\_ports (skrf.network.Network attribute), 63

## O

one\_port() (in module skrf.calibration.calibrationAlgorithms), 173  
 one\_port\_nls() (in module skrf.calibration.calibrationAlgorithms), 174  
 open() (skrf.media.cpw.CPW method), 236  
 open() (skrf.media.distributedCircuit.DistributedCircuit method), 208  
 open() (skrf.media.freespace.Freespace method), 250  
 open() (skrf.media.media.Media method), 194  
 open() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 223  
 output\_from\_cal (skrf.calibration.calibration.Calibration attribute), 168, 179

## P

parameterized\_self\_calibration() (in module skrf.calibration.calibrationAlgorithms), 175  
 parameterized\_self\_calibration\_nls() (in module skrf.calibration.calibrationAlgorithms), 176  
 passivity (skrf.network.Network attribute), 63  
 plot\_a\_arcl() (skrf.network.Network method), 77  
 plot\_a\_arcl\_unwrap() (skrf.network.Network method), 78  
 plot\_a\_complex() (skrf.network.Network method), 78  
 plot\_a\_db() (skrf.network.Network method), 79  
 plot\_a\_deg() (skrf.network.Network method), 80  
 plot\_a\_deg\_unwrap() (skrf.network.Network method), 81  
 plot\_a\_im() (skrf.network.Network method), 81  
 plot\_a\_mag() (skrf.network.Network method), 82  
 plot\_a\_polar() (skrf.network.Network method), 83  
 plot\_a\_rad() (skrf.network.Network method), 84  
 plot\_a\_rad\_unwrap() (skrf.network.Network method), 84  
 plot\_a\_re() (skrf.network.Network method), 85  
 plot\_coefs\_db() (skrf.calibration.calibration.Calibration method), 170, 182  
 plot\_complex\_polar() (in module skrf.plotting), 142  
 plot\_complex\_rectangular() (in module skrf.plotting), 141  
 plot\_errors() (skrf.calibration.calibration.Calibration method), 170, 182  
 plot\_it\_all() (skrf.network.Network method), 86  
 plot\_logsigma() (skrf.networkSet.NetworkSet method), 135  
 plot\_passivity() (skrf.network.Network method), 86  
 plot\_polar() (in module skrf.plotting), 140  
 plot\_rectangular() (in module skrf.plotting), 140  
 plot\_residuals() (skrf.calibration.calibration.Calibration method), 171, 183  
 plot\_residuals\_db() (skrf.calibration.calibration.Calibration method), 171, 183  
 plot\_residuals\_mag() (skrf.calibration.calibration.Calibration method), 171, 183  
 plot\_residuals\_smith() (skrf.calibration.calibration.Calibration method), 171, 183  
 plot\_s\_arcl() (skrf.network.Network method), 86  
 plot\_s\_arcl\_unwrap() (skrf.network.Network method), 87  
 plot\_s\_complex() (skrf.network.Network method), 88  
 plot\_s\_db() (skrf.network.Network method), 89  
 plot\_s\_deg() (skrf.network.Network method), 89  
 plot\_s\_deg\_unwrap() (skrf.network.Network method), 90  
 plot\_s\_im() (skrf.network.Network method), 91  
 plot\_s\_mag() (skrf.network.Network method), 92  
 plot\_s\_polar() (skrf.network.Network method), 92  
 plot\_s\_rad() (skrf.network.Network method), 93  
 plot\_s\_rad\_unwrap() (skrf.network.Network method), 94  
 plot\_s\_re() (skrf.network.Network method), 95  
 plot\_s\_smith() (skrf.network.Network method), 95  
 plot\_smith() (in module skrf.plotting), 139

- plot\_uncertainty\_bounds\_component()  
     (skrf.networkSet.NetworkSet method), 135  
 plot\_uncertainty\_bounds\_s()  
     (skrf.networkSet.NetworkSet method), 136  
 plot\_uncertainty\_bounds\_s\_db()  
     (skrf.networkSet.NetworkSet method), 136  
 plot\_uncertainty\_decomposition()  
     (skrf.networkSet.NetworkSet method), 136  
 plot\_uncertainty\_per\_standard()  
     (skrf.calibration.calibration.Calibration  
     method), 171, 183  
 plot\_y\_arcl() (skrf.network.Network method), 96  
 plot\_y\_arcl\_unwrap() (skrf.network.Network method),  
     97  
 plot\_y\_complex() (skrf.network.Network method), 98  
 plot\_y\_db() (skrf.network.Network method), 98  
 plot\_y\_deg() (skrf.network.Network method), 99  
 plot\_y\_deg\_unwrap() (skrf.network.Network method),  
     100  
 plot\_y\_im() (skrf.network.Network method), 101  
 plot\_y\_mag() (skrf.network.Network method), 101  
 plot\_y\_polar() (skrf.network.Network method), 102  
 plot\_y\_rad() (skrf.network.Network method), 103  
 plot\_y\_rad\_unwrap() (skrf.network.Network method),  
     104  
 plot\_y\_re() (skrf.network.Network method), 104  
 plot\_z\_arcl() (skrf.network.Network method), 105  
 plot\_z\_arcl\_unwrap() (skrf.network.Network method),  
     106  
 plot\_z\_complex() (skrf.network.Network method), 107  
 plot\_z\_db() (skrf.network.Network method), 107  
 plot\_z\_deg() (skrf.network.Network method), 108  
 plot\_z\_deg\_unwrap() (skrf.network.Network method),  
     109  
 plot\_z\_im() (skrf.network.Network method), 110  
 plot\_z\_mag() (skrf.network.Network method), 110  
 plot\_z\_polar() (skrf.network.Network method), 111  
 plot\_z\_rad() (skrf.network.Network method), 112  
 plot\_z\_rad\_unwrap() (skrf.network.Network method),  
     113  
 plot\_z\_re() (skrf.network.Network method), 113  
 pna\_csv\_2\_ntwks() (in module skrf.io.csv), 166  
 propagation\_constant (skrf.media.cpw.CPW attribute),  
     229  
 propagation\_constant (skrf.media.distributedCircuit.DistributedCircuit  
     attribute), 200  
 propagation\_constant (skrf.media.freespace.Freespace at-  
     tribute), 242  
 propagation\_constant (skrf.media.media.Media attribute),  
     186  
 propagation\_constant (skrf.media.rectangularWaveguide.RectangularWaveguide  
     attribute), 215  
 propagation\_impedance\_2\_distributed\_circuit() (in mod-  
     ule skrf.tlineFunctions), 154
- ## R
- radian\_2\_degree() (in module skrf.mathFunctions), 145  
 read() (in module skrf.io.general), 158  
 read() (skrf.network.Network method), 114  
 read\_all() (in module skrf.io.general), 159  
 read\_pna\_csv() (in module skrf.io.csv), 165  
 read\_touchstone() (skrf.network.Network method), 115  
 RectangularWaveguide (class in  
     skrf.media.rectangularWaveguide), 213  
 reflection\_coefficient\_2\_input\_impedance() (in module  
     skrf.tlineFunctions), 152  
 reflection\_coefficient\_2\_input\_impedance\_at\_theta() (in  
     module skrf.tlineFunctions), 152  
 reflection\_coefficient\_at\_theta() (in module  
     skrf.tlineFunctions), 151  
 renumber() (skrf.network.Network method), 115  
 resample() (skrf.network.Network method), 115, 120  
 residual\_ntwks (skrf.calibration.calibration.Calibration  
     attribute), 168, 180  
 residuals (skrf.calibration.calibration.Calibration at-  
     tribute), 168, 180  
 resistor() (skrf.media.cpw.CPW method), 236  
 resistor() (skrf.media.distributedCircuit.DistributedCircuit  
     method), 208  
 resistor() (skrf.media.freespace.Freespace method), 250  
 resistor() (skrf.media.media.Media method), 194  
 resistor() (skrf.media.rectangularWaveguide.RectangularWaveguide  
     method), 223  
 run() (skrf.calibration.calibration.Calibration method),  
     171, 183
- ## S
- s (skrf.network.Network attribute), 64  
 s11 (skrf.network.Network attribute), 64  
 s12 (skrf.network.Network attribute), 64  
 s21 (skrf.network.Network attribute), 64  
 s22 (skrf.network.Network attribute), 64  
 s2t() (in module skrf.network), 127  
 s2y() (in module skrf.network), 126  
 s2z() (in module skrf.network), 126  
 s\_arcl (skrf.network.Network attribute), 64  
 s\_arcl\_unwrap (skrf.network.Network attribute), 65  
 s\_db (skrf.network.Network attribute), 65  
 s\_deg (skrf.network.Network attribute), 65  
 s\_deg\_unwrap (skrf.network.Network attribute), 65  
 s\_im (skrf.network.Network attribute), 65  
 s\_mag (skrf.network.Network attribute), 65  
 s\_rad (skrf.network.Network attribute), 66  
 s\_rad\_unwrap (skrf.network.Network attribute), 66  
 s\_re (skrf.network.Network attribute), 66  
 save\_all\_figs() (in module skrf.plotting), 143  
 save\_sesh() (in module skrf.io.general), 162  
 scalar2Complex() (in module skrf.mathFunctions), 146

- set\_wise\_function() (skrf.networkSet.NetworkSet method), 137
- short() (skrf.media.cpw.CPW method), 237
- short() (skrf.media.distributedCircuit.DistributedCircuit method), 209
- short() (skrf.media.freespace.Freespace method), 251
- short() (skrf.media.media.Media method), 194
- short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 223
- shunt() (skrf.media.cpw.CPW method), 237
- shunt() (skrf.media.distributedCircuit.DistributedCircuit method), 209
- shunt() (skrf.media.freespace.Freespace method), 251
- shunt() (skrf.media.media.Media method), 195
- shunt() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 224
- shunt\_capacitor() (skrf.media.cpw.CPW method), 237
- shunt\_capacitor() (skrf.media.distributedCircuit.DistributedCircuit method), 209
- shunt\_capacitor() (skrf.media.freespace.Freespace method), 251
- shunt\_capacitor() (skrf.media.media.Media method), 195
- shunt\_capacitor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 224
- shunt\_delay\_load() (skrf.media.cpw.CPW method), 238
- shunt\_delay\_load() (skrf.media.distributedCircuit.DistributedCircuit method), 210
- shunt\_delay\_load() (skrf.media.freespace.Freespace method), 252
- shunt\_delay\_load() (skrf.media.media.Media method), 195
- shunt\_delay\_load() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 224
- shunt\_delay\_open() (skrf.media.cpw.CPW method), 238
- shunt\_delay\_open() (skrf.media.distributedCircuit.DistributedCircuit method), 210
- shunt\_delay\_open() (skrf.media.freespace.Freespace method), 252
- shunt\_delay\_open() (skrf.media.media.Media method), 196
- shunt\_delay\_open() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 225
- shunt\_delay\_short() (skrf.media.cpw.CPW method), 238
- shunt\_delay\_short() (skrf.media.distributedCircuit.DistributedCircuit method), 210
- shunt\_delay\_short() (skrf.media.freespace.Freespace method), 252
- shunt\_delay\_short() (skrf.media.media.Media method), 196
- shunt\_delay\_short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 225
- shunt\_inductor() (skrf.media.cpw.CPW method), 239
- shunt\_inductor() (skrf.media.distributedCircuit.DistributedCircuit method), 211
- shunt\_inductor() (skrf.media.freespace.Freespace method), 253
- shunt\_inductor() (skrf.media.media.Media method), 196
- shunt\_inductor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 225
- signature() (skrf.networkSet.NetworkSet method), 137
- skin\_depth() (in module skrf.tlineFunctions), 155
- skrf.calibration (module), 166
- skrf.calibration.calibration (module), 166
- skrf.calibration.calibrationAlgorithms (module), 173
- skrf.calibration.calibrationFunctions (module), 177
- skrf.constants (module), 155
- skrf.frequency (module), 53
- skrf.io (module), 158
- skrf.io.csv (module), 165
- skrf.io.general (module), 158
- skrf.io.touchstone (module), 162
- skrf.mathFunctions (module), 144
- skrf.media (module), 185
- skrf.network (module), 57
- skrf.networkSet (module), 132
- skrf.plotting (module), 138
- skrf.util (module), 156
- smith() (in module skrf.plotting), 138
- smith() (skrf.frequency.Frequency attribute), 55
- splitter() (skrf.media.cpw.CPW method), 239
- splitter() (skrf.media.distributedCircuit.DistributedCircuit method), 211
- splitter() (skrf.media.freespace.Freespace method), 253
- splitter() (skrf.media.media.Media method), 197
- splitter() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 226
- sqrt\_phase\_unwrap() (in module skrf.mathFunctions), 145
- start (skrf.frequency.Frequency attribute), 55
- std\_s\_db (skrf.networkSet.NetworkSet attribute), 134
- step (skrf.frequency.Frequency attribute), 55
- stitch() (in module skrf.network), 122
- stop (skrf.frequency.Frequency attribute), 55
- stopband\_width() (in module skrf.tlineFunctions), 155

## T

- t (skrf.network.Network attribute), 66
- t2s() (in module skrf.network), 130
- t2y() (in module skrf.network), 131
- t2z() (in module skrf.network), 130
- tee() (skrf.media.cpw.CPW method), 240
- tee() (skrf.media.distributedCircuit.DistributedCircuit method), 211
- tee() (skrf.media.freespace.Freespace method), 253
- tee() (skrf.media.media.Media method), 197
- tee() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 226

- theta() (in module skrf.tlineFunctions), 148
- theta\_2\_d() (skrf.media.cpw.CPW method), 240
- theta\_2\_d() (skrf.media.distributedCircuit.DistributedCircuit method), 212
- theta\_2\_d() (skrf.media.freespace.Freespace method), 254
- theta\_2\_d() (skrf.media.media.Media method), 197
- theta\_2\_d() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 226
- thru() (skrf.media.cpw.CPW method), 240
- thru() (skrf.media.distributedCircuit.DistributedCircuit method), 212
- thru() (skrf.media.freespace.Freespace method), 254
- thru() (skrf.media.media.Media method), 198
- thru() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 227
- total\_error() (skrf.calibration.calibration.Calibration method), 172, 184
- Touchstone (class in skrf.io.touchstone), 162
- Ts (skrf.calibration.calibration.Calibration attribute), 167, 179
- two\_port() (in module skrf.calibration.calibrationAlgorithms), 174
- type (skrf.calibration.calibration.Calibration attribute), 168, 180
- ## U
- unbiased\_error() (skrf.calibration.calibration.Calibration method), 172, 184
- uncertainty\_ntwk\_triplet() (skrf.networkSet.NetworkSet method), 137
- uncertainty\_per\_standard() (skrf.calibration.calibration.Calibration method), 172, 184
- unit (skrf.frequency.Frequency attribute), 55
- unit\_dict (skrf.frequency.Frequency attribute), 55
- unterminate\_switch\_terms() (in module skrf.calibration.calibrationAlgorithms), 177
- unwrap\_rad() (in module skrf.mathFunctions), 145
- ## W
- w (skrf.frequency.Frequency attribute), 56
- white\_gaussian\_polar() (skrf.media.cpw.CPW method), 241
- white\_gaussian\_polar() (skrf.media.distributedCircuit.DistributedCircuit method), 212
- white\_gaussian\_polar() (skrf.media.freespace.Freespace method), 254
- white\_gaussian\_polar() (skrf.media.media.Media method), 198
- white\_gaussian\_polar() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 227
- write() (in module skrf.io.general), 160
- write() (skrf.calibration.calibration.Calibration method), 173, 185
- write() (skrf.network.Network method), 116
- write() (skrf.networkSet.NetworkSet method), 138
- write\_all() (in module skrf.io.general), 161
- write\_csv() (skrf.media.cpw.CPW method), 241
- write\_csv() (skrf.media.distributedCircuit.DistributedCircuit method), 213
- write\_csv() (skrf.media.freespace.Freespace method), 255
- write\_csv() (skrf.media.media.Media method), 199
- write\_csv() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 228
- write\_touchstone() (skrf.network.Network method), 116
- ## Y
- Y (skrf.media.distributedCircuit.DistributedCircuit attribute), 200
- Y (skrf.media.freespace.Freespace attribute), 242
- y (skrf.network.Network attribute), 66
- y2s() (in module skrf.network), 129
- y2t() (in module skrf.network), 130
- y2z() (in module skrf.network), 129
- y\_arcl (skrf.network.Network attribute), 67
- y\_arcl\_unwrap (skrf.network.Network attribute), 67
- y\_db (skrf.network.Network attribute), 67
- y\_deg (skrf.network.Network attribute), 67
- y\_deg\_unwrap (skrf.network.Network attribute), 67
- y\_im (skrf.network.Network attribute), 68
- y\_mag (skrf.network.Network attribute), 68
- y\_rad (skrf.network.Network attribute), 68
- y\_rad\_unwrap (skrf.network.Network attribute), 68
- y\_re (skrf.network.Network attribute), 68
- ## Z
- Z (skrf.media.distributedCircuit.DistributedCircuit attribute), 200
- Z (skrf.media.freespace.Freespace attribute), 242
- z (skrf.network.Network attribute), 68
- z0 (skrf.media.cpw.CPW attribute), 229
- z0 (skrf.media.distributedCircuit.DistributedCircuit attribute), 201
- z0 (skrf.media.freespace.Freespace attribute), 243
- z0 (skrf.media.media.Media attribute), 187
- z0 (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 215
- z0 (skrf.network.Network attribute), 69
- Z0() (skrf.media.cpw.CPW method), 230
- Z0() (skrf.media.distributedCircuit.DistributedCircuit method), 202
- Z0() (skrf.media.freespace.Freespace method), 244
- Z0() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 216
- z2s() (in module skrf.network), 127

`z2t()` (in module `skrf.network`), 128  
`z2y()` (in module `skrf.network`), 128  
`z_arcl` (`skrf.network.Network` attribute), 69  
`z_arcl_unwrap` (`skrf.network.Network` attribute), 69  
`z_db` (`skrf.network.Network` attribute), 69  
`z_deg` (`skrf.network.Network` attribute), 69  
`z_deg_unwrap` (`skrf.network.Network` attribute), 70  
`z_im` (`skrf.network.Network` attribute), 70  
`z_mag` (`skrf.network.Network` attribute), 70  
`z_rad` (`skrf.network.Network` attribute), 70  
`z_rad_unwrap` (`skrf.network.Network` attribute), 70  
`z_re` (`skrf.network.Network` attribute), 70  
`zl_2_Gamma0()` (in module `skrf.tlineFunctions`), 148  
`zl_2_Gamma_in()` (in module `skrf.tlineFunctions`), 149  
`zl_2_zin()` (in module `skrf.tlineFunctions`), 149