scikit-rf Documentation

Release dev

Alex Arsenovic

January 21, 2013

CONTENTS

1 Tutorials **3**
- 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

2 Examples **47**
- 2.1 Visualizing a Single Stub Matching Network . . . 47
- 2.2 One-Port Calibration . . . 51

3 Reference **53**
- 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`) . . . 165
- 3.11 media (`skrf.media`) . . . 184
- 3.12 vi (`skrf.vi`) . . . 254
- 3.13 Indices and tables . . . 267

Python Module Index **269**

Index **271**

This documentation is also available in pdf form: scikit-rf.pdf

CHAPTER

ONE

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 *ditributions* 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 [1]: 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 unsuccesful. 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 (>=2.6) http://www.python.org/
- matplotlib (aka pylab) http://matplotlib.sourceforge.net/
- numpy http://numpy.scipy.org/
- scipy http://www.scipy.org/ (provides tons of good stuff, check it out)

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 Compoents

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 [8]: 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 [9]: 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 [10]: ring_slot
Out[10]: 2-Port Network: 'ring slot',  75-110 GHz, 201 pts, z0=[ 50.+0.j  50.+0.j]
```

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

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

All of the network parameters are complex `numpy.ndarray` 's of shape *FxNxN*, 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 found in the `Network` docstring. If you are using IPython, then these properties and methods can be 'tabbed' out on the command line.

```
In [11]: 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 Networks stored in the `data` module.

```
In [12]: short = rf.data.wr2p2_short

In [13]: delayshort = rf.data.wr2p2_delayshort

In [14]: short - delayshort
Out[14]: 1-Port Network: 'wr2p2,short',  330-500 GHz, 201 pts, z0=[ 50.+0.j]

In [15]: short + delayshort
Out[15]: 1-Port Network: 'wr2p2,short',  330-500 GHz, 201 pts, z0=[ 50.+0.j]
```

Cascading and De-embedding

Cascading and de-embeding 2-port Networks can also be done though 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 [16]: short = rf.data.wr2p2_short

In [17]: line = rf.data.wr2p2_line

In [18]: 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 [19]: 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.

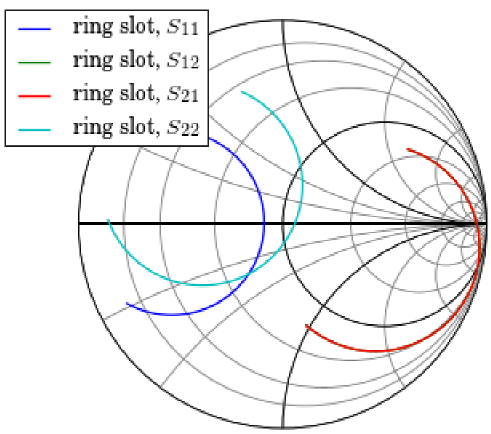
1.2.3 Plotting

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 [20]: 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 [21]: rf.read_all('../skrf/data/', contains='ro')
Out[21]:
{'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 [22]: ro_dict = rf.read_all('../skrf/data/', contains='ro')

In [23]: ro_ns = rf.NetworkSet(ro_dict, name='ro set') #name is optional
```

```
In [24]: ro_ns
Out[24]: 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

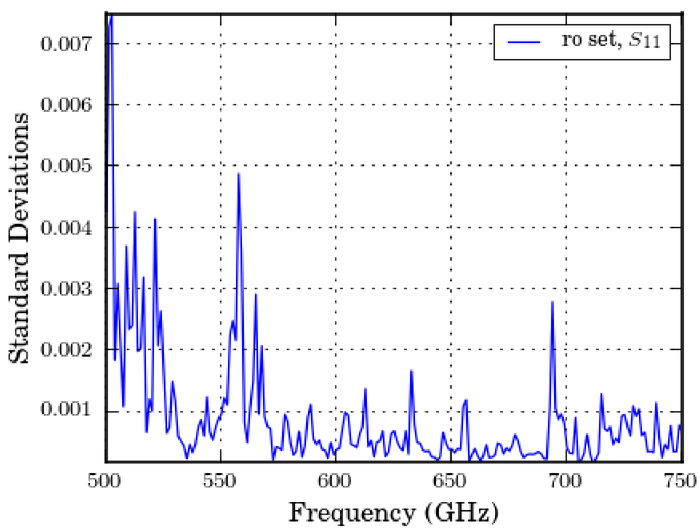
```
In [25]: ro_ns.mean_s
Out[25]: 1-Port Network: 'ro set',  500-750 GHz, 201 pts, z0=[ 50.+0.j]
```

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

```
In [26]: ro_ns.std_s
Out[26]: 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,

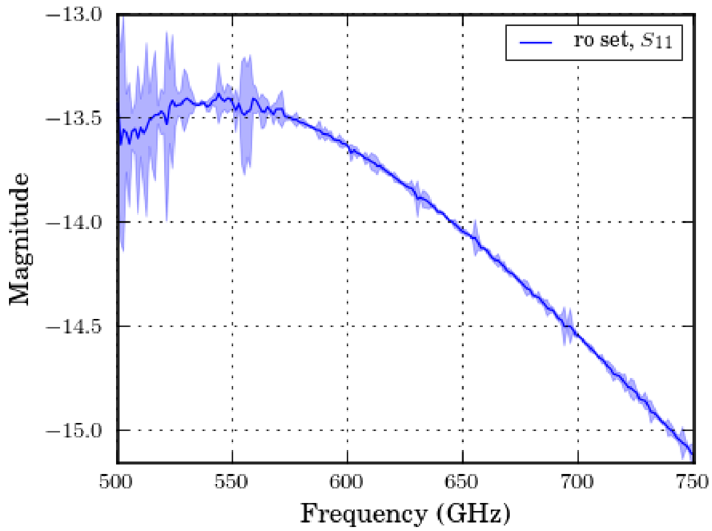
```
In [27]: figure();

In [28]: 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 [29]: figure();

In [30]: 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 [31]: from skrf.vi import vna

In [32]: 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 [33]: dut_1 = my_vna.s11

In [34]: dut_2 = my_vna.s21

In [35]: 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**‘sdefault 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 relevent such as,

- calibration algorithm
- enforce eciprocity 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 neccesary properties such as `propagation constant` and `characteristic impedance`, that are needed to generate microwave circuits.

The basic usage looks something like this,

```
In [36]: import skrf as rf

In [37]: freq = rf.Frequency(75,110,101,'ghz')
```

```
In [38]: cpw = rf.media.CPW(freq, w=10e-6, s=5e-6, ep_r=10.6)

In [39]: cpw.line(100*1e-6, name = '100um line')
Out[39]: 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 cabilities 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 Compoents

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 dicontinuity effects),

```
In [40]: cpw.short(name = 'short')
Out[40]: 1-Port Network: 'short',  75-110 GHz, 101 pts, z0=[ 50.06074662+0.j]
```

Or to create a 90° section of cpw line,

```
In [41]: cpw.line(d=90,unit='deg', name='line')
Out[41]: 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)
```

Seen `Network.__init__()` for more informaition 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 *FxNxN*, 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+0.4578448j , -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
...
```

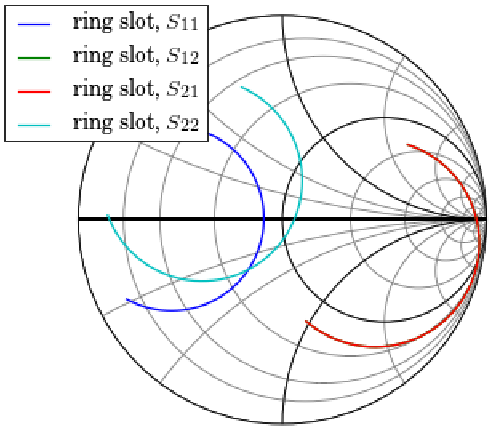
Note: Although this tutorial focuses on s-parametes, other network representations such as Impedance (`Network.z`) and Admittance Parameters (`Network.y`) are available as well, see Impedance and Admittance Parameters .

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
- ...

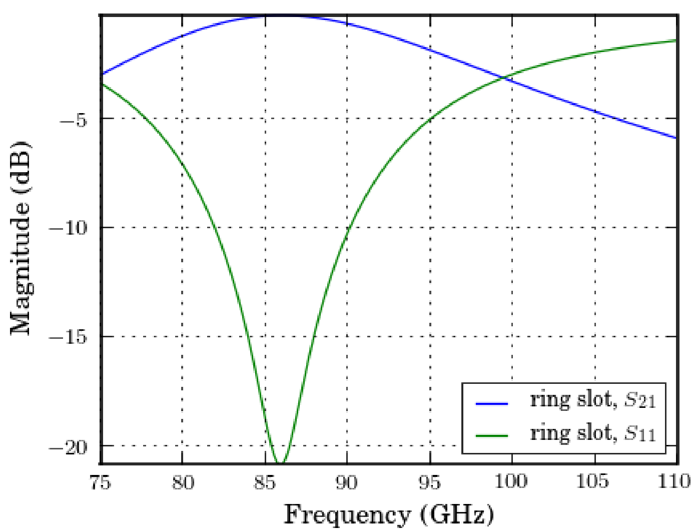
To plot all four s-parameters of the `ring_slot` on the Smith Chart.

```
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

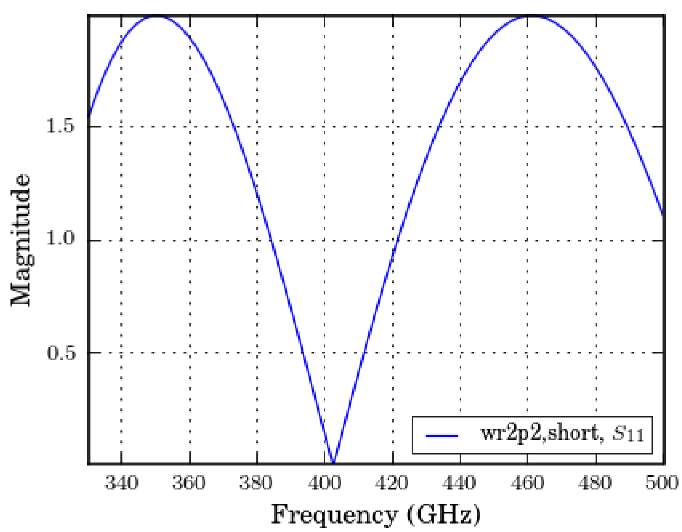
Linear Operations

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]
```

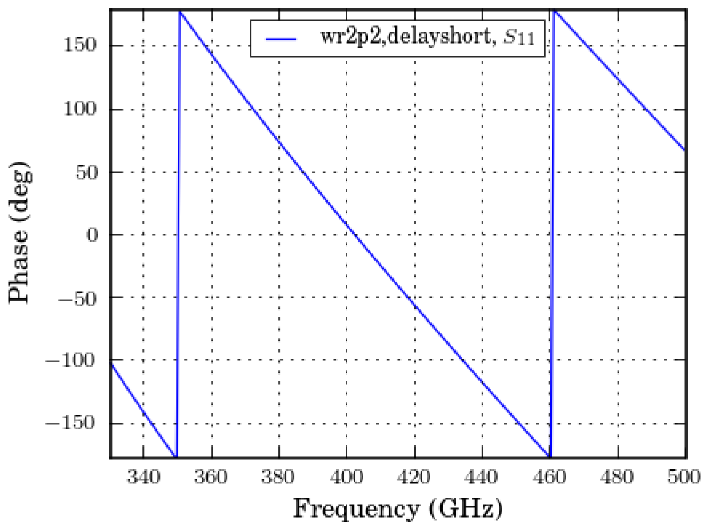
All of these operations return `Network` types, so the methods and properties of a `Network` are available on the result. For example, to plot the complex difference between `short` and `delay_short`,

```
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 ais 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([[[ 0.74019959+0.j]],

       [[ 0.73802816+0.j]],

       [[ 0.50144126+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-embeding 2-port Networks can also be done though 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 [1], 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 *1* of the tee, to port *0* of the delay short,

```
In [1]: terminated_tee = rf.connect(tee,1,delayshort,0)
```

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)
    438
    439         if isinstance(other, Network):
--> 440             self.__compatable_for_scalar_operation_test(other)
    441             result.s = self.s + other.s
    442         else:

/home/alex/data/docs/code/path/skrf/network.pyc in __compatable_for_scalar_operation_test(self, other
    564         '''
    565         if other.frequency  != self.frequency:
--> 566             raise IndexError('Networks must have same frequency. See `Network.interpolate`')
    567
    568         if other.s.shape != self.s.shape:
```

[1] 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 Netwoks 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()`,

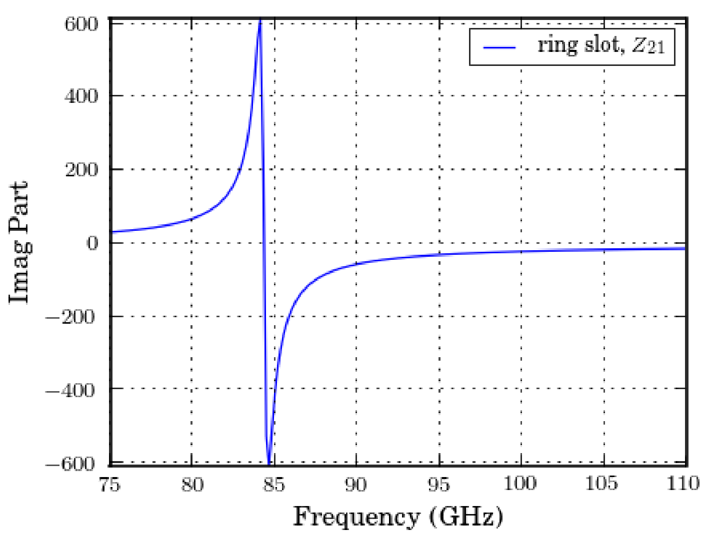
```
In [1]: rf.write('dict_o_ntwk.p', dict_o_ntwks)

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 represenations 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_z_im()`
- ...

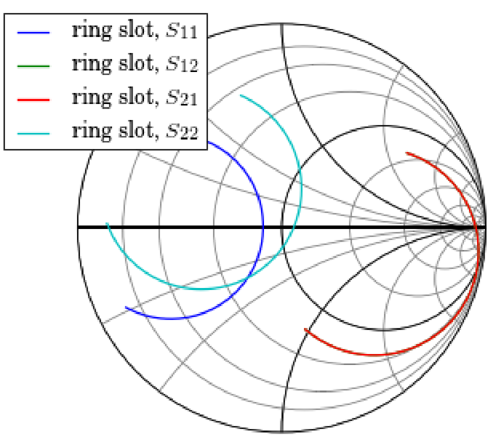
Step-by-step examples of how to create and customize plots are given below.

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()
```

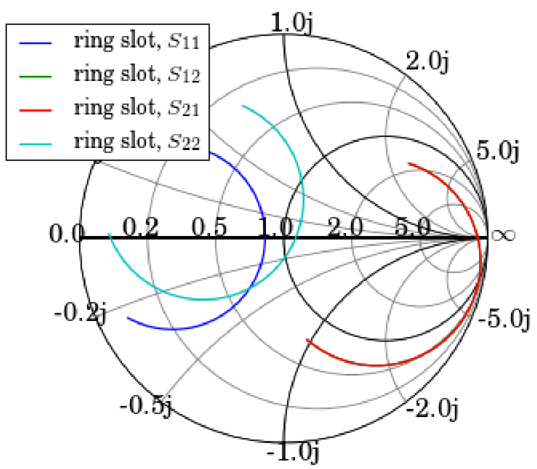


Note: If you dont see any plots after issuing these commands, then you may not have started ipython with the `--pylab` flag. Try `from pylab import *` to import the matplotlib commands and `ion()` to turn on interactive plotting. See this page , for more info on ipython's *pylab* mode.

Note: Why do my plots look different? See Formating Plots

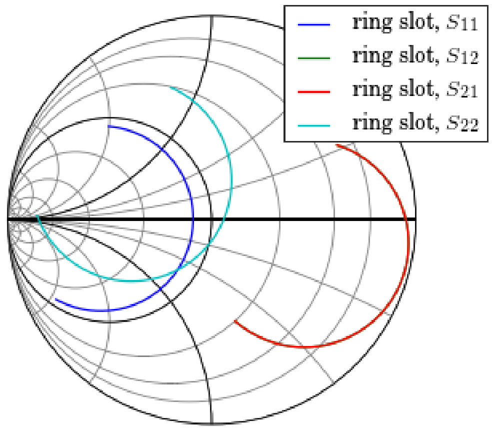
The smith chart can be drawn with some impedance values labeled through the `draw_labels` argument.

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

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



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

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

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



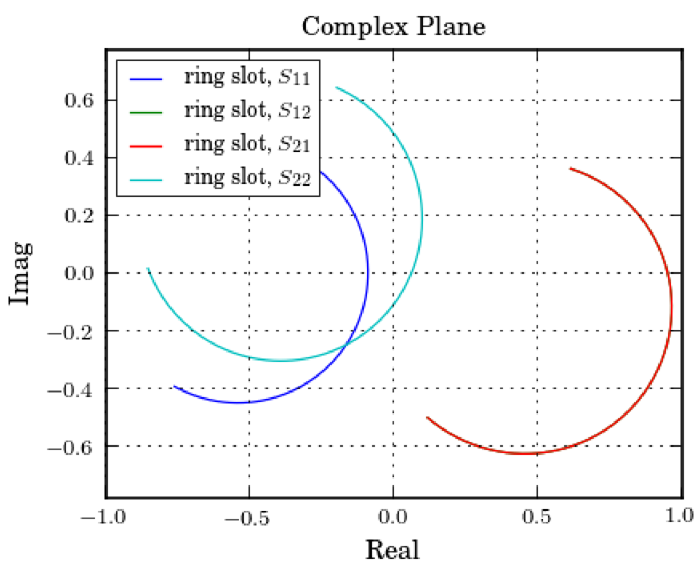
See `smith()` for more info on customizing the Smith Chart.

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();
```

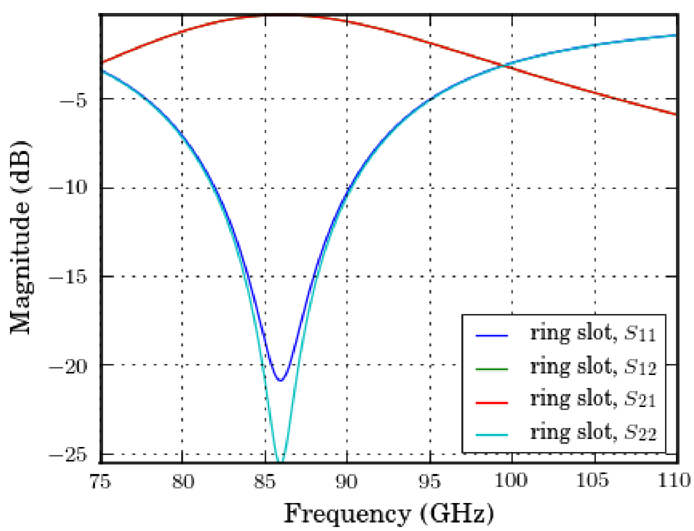


1.4.3 Rectangular Plots

Log-Magnitude

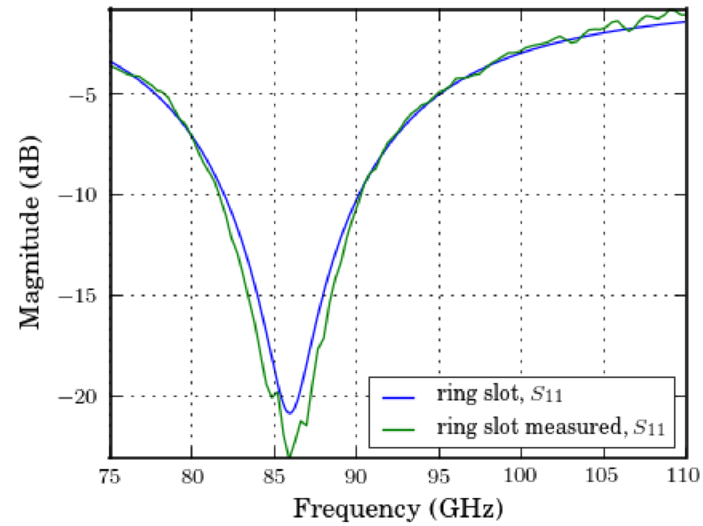
Scalar components of the complex network parameters can be plotted vs frequency as well. To plot the log-magnitude of the s-parameters vs. frequency,

```
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 indecies `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
```

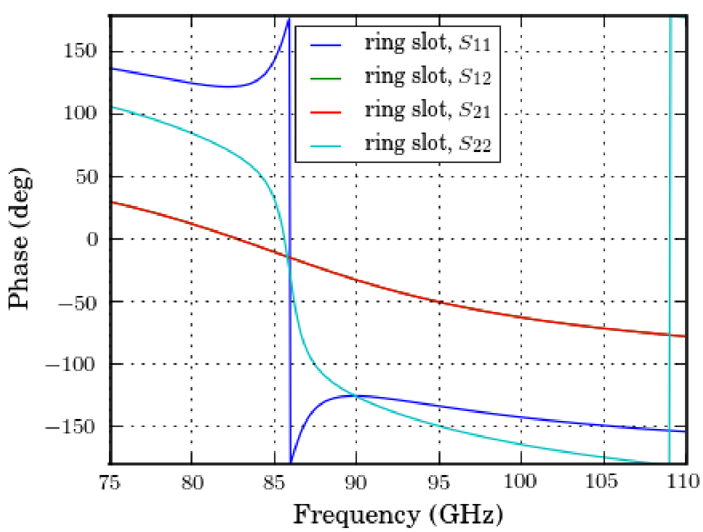


See Customizing Plots for more information on customization.

Phase

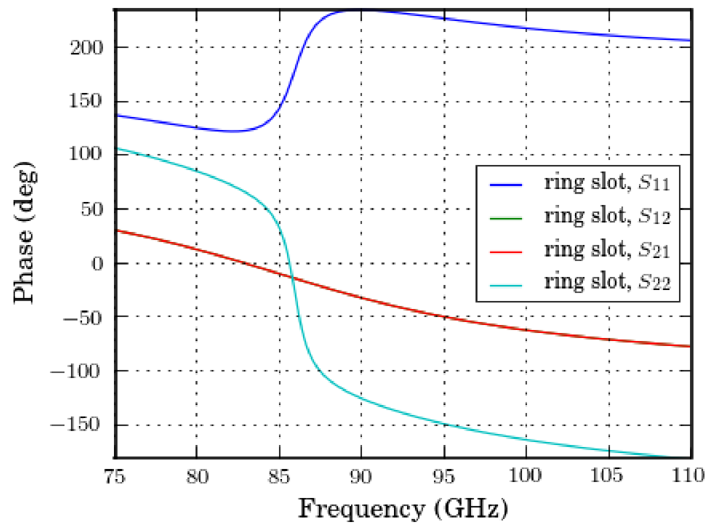
Plot phase,

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

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



Or unwrapped phase,

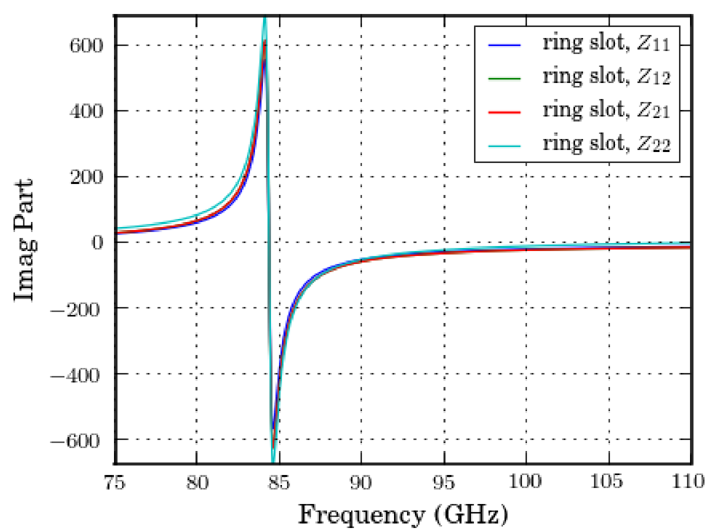
```
In [1]: figure();

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



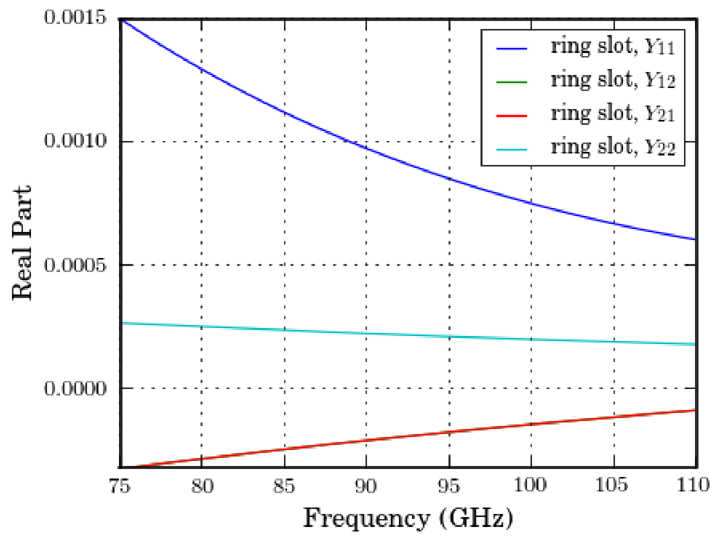
Impedance, Admittance

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

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

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



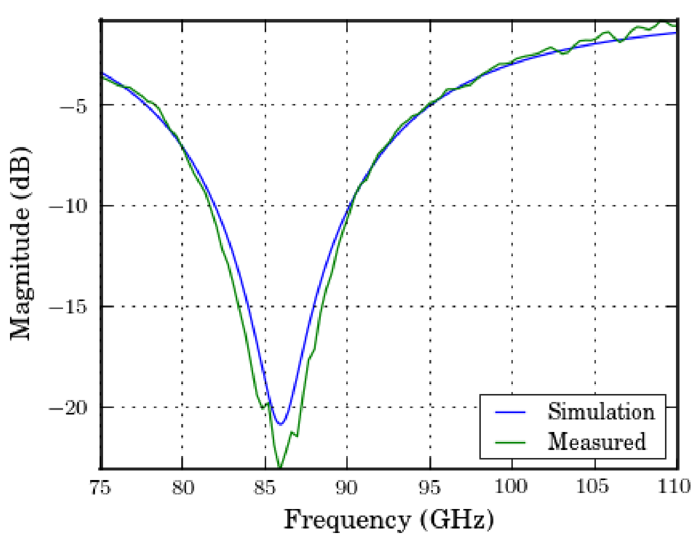

```
In [1]: figure();

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 overidden 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')
```

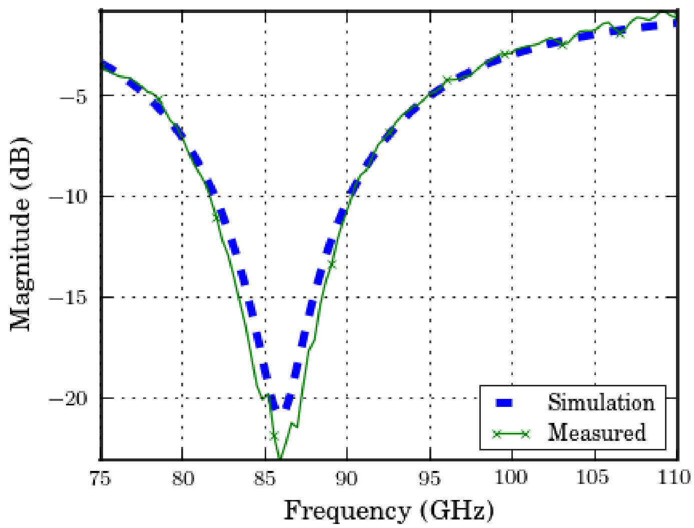


The frequency unit used on the x-axis is automatically filled in from the Networks `frequency` attribute. To change the label, change the frequency's `unit`.

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

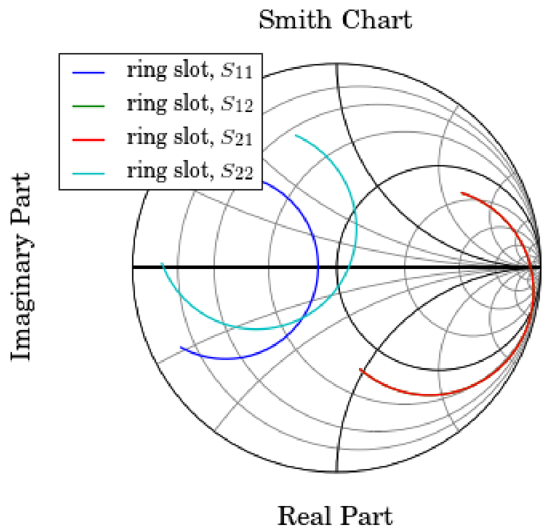
Other key word arguments given to the plotting methods are passed through to the matplotlib `plot()` function.

```
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:

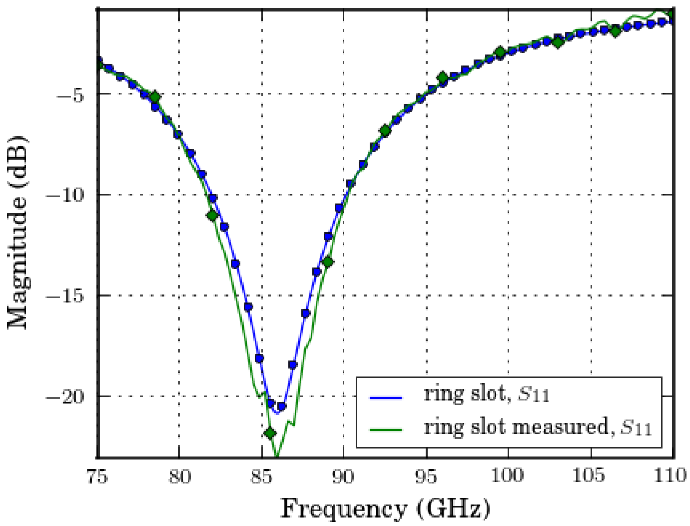
```
>>> rf.save_all_figs('.', format=['eps','pdf'])
./WR-10 Ringslot Array Simulated vs Measured.eps
./WR-10 Ringslot Array Simulated vs Measured.pdf
```

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,*.s1p`,

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

In [2]: ls ../skrf/data/ro*
../skrf/data/ro,1.s1p  ../skrf/data/ro,2.s1p  ../skrf/data/ro,3.s1p
```

The files `ro,1.s1p` , `ro,2.s1p`, ... 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()`

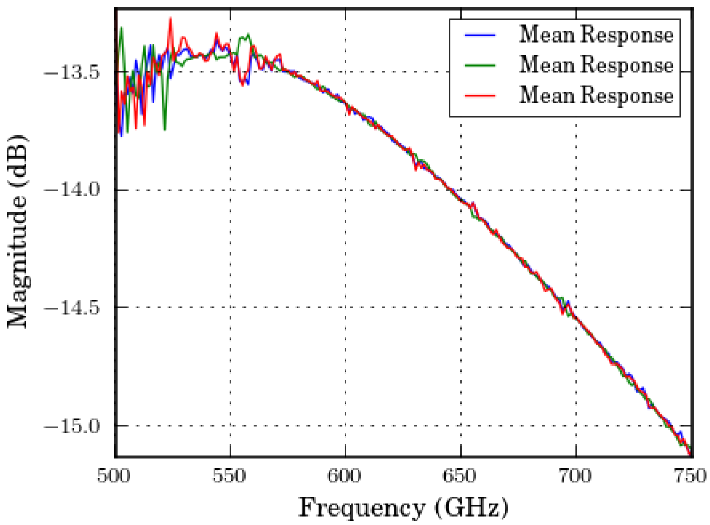
1.5.2 Accesing Network Methods

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 ploting methods).

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



1.5.3 Statistical Properties

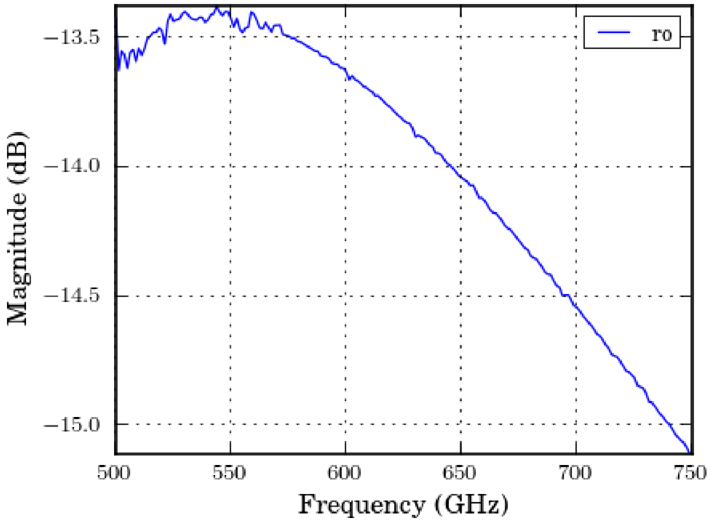
Statistical quantities can be calculated by accessing properties of the NetworkSet. For example, to calculate the complex average of the set, access the mean_s property

```
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.

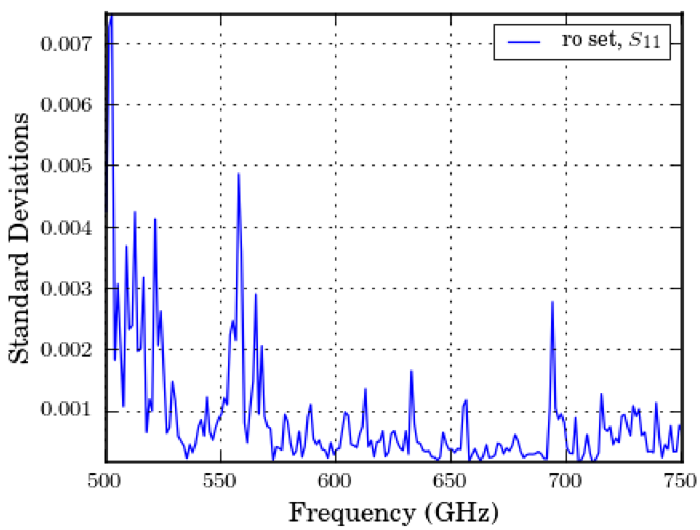
The naming convention of the statistical operator properties are *NetworkSet.function_parameter*, where *function* is the name of the statistical function, and *parameter* is the Network parameter to operate on. These methods return a Network object, so they can be saved or plotted in the same way as 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')
```



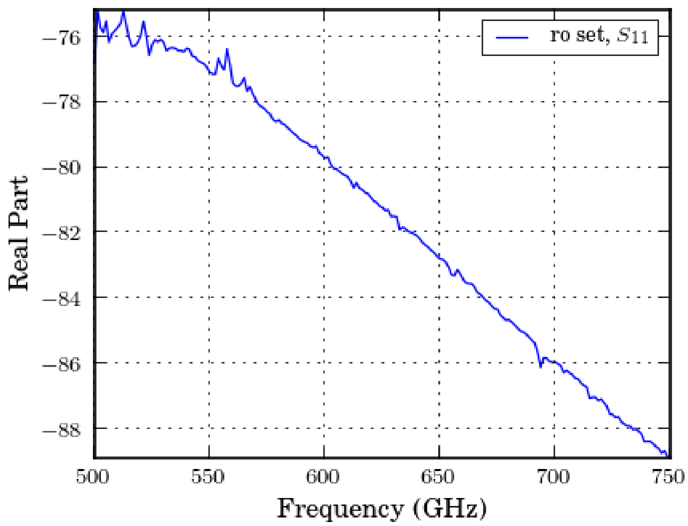
Or to plot the standard deviation of the complex s-parameters,

```
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,

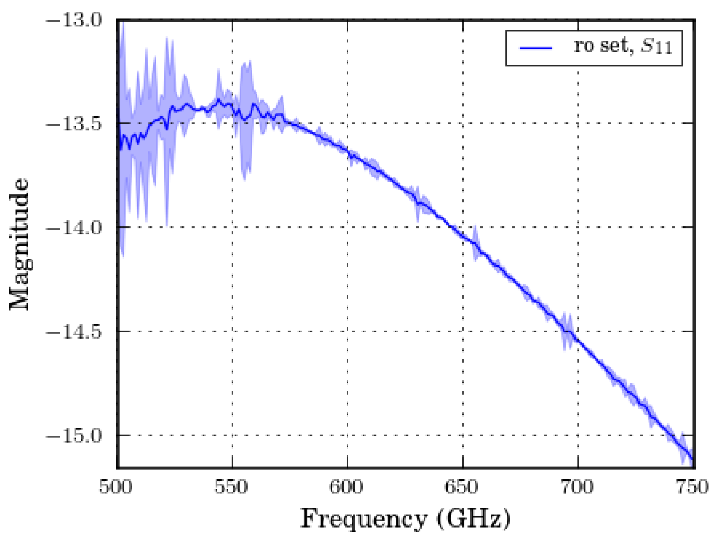
```
In [1]: figure();

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

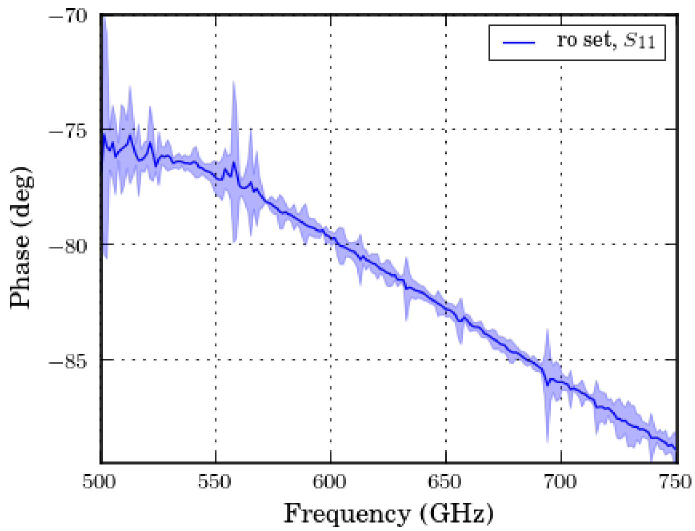


1.5.4 Plotting Uncertainty Bounds

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 ccomplished 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 relevent such as,

- calibration algorithm
- enforce eciprocity 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.s1p'),
        rf.Network('ideal/open.s1p'),
        rf.Network('ideal/load.s1p'),
        ]

# a list of Network types, holding 'measured' responses
my_measured = [\
        rf.Network('measured/short.s1p'),
        rf.Network('measured/open.s1p'),
        rf.Network('measured/load.s1p'),
        ]

## 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.s1p')
dut_caled = cal.apply_cal(dut)

# plot results
dut_caled.plot_s_db()
# save results
dut_caled.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 [2], 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 [3] , 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.

[2] 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

[3] 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 (S21=S12=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_caled = cal.apply_cal(dut)

# plot results
dut_caled.plot_s_db()
# save results
dut_caled.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.s1p')
load = rf.Network('ideals/load.s1p')
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 neccesary 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 cabilities 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 repitously provide frequency information for each new network created.

Coplanar Waveguide

Here is an example of how to initialize a coplanar waveguide [4] 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.

[4] 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 specifc 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.

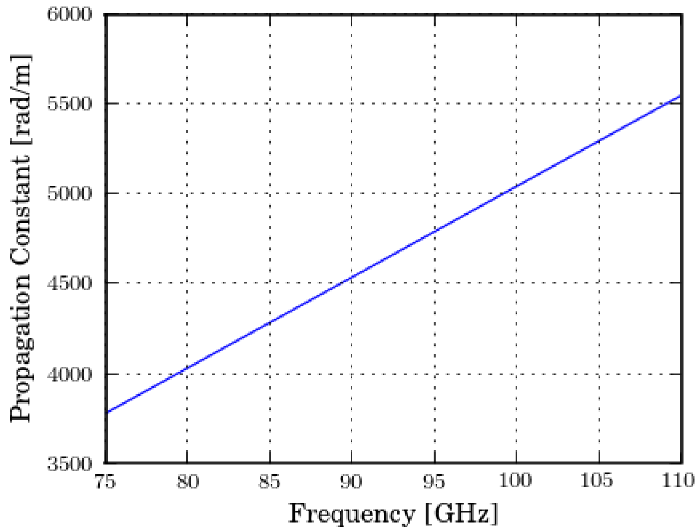
1.8.3 Working with Media's

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])
```

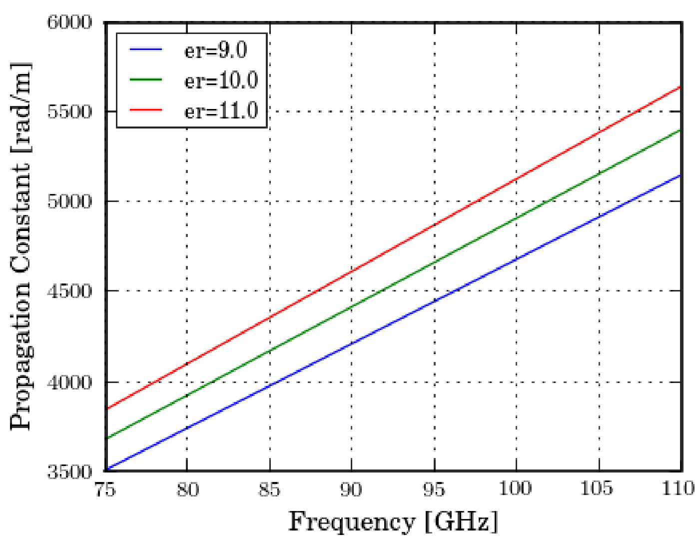
As an example, plot the cpw's propagation constant vs frequency.

```
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 frequenctly, it may make sense to use a function to create the circuit. This can be done most quickly using lamba

```
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(med, d0, d1):
   ...:     return med.line(d0,'deg')**med.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 somthing 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,zl=100))

# single stub circuit generator function
In [3]: def shunt_stub(med, d0, d1):
   ...:     return med.line(d0,'deg')**med.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

CHAPTER

TWO

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
```

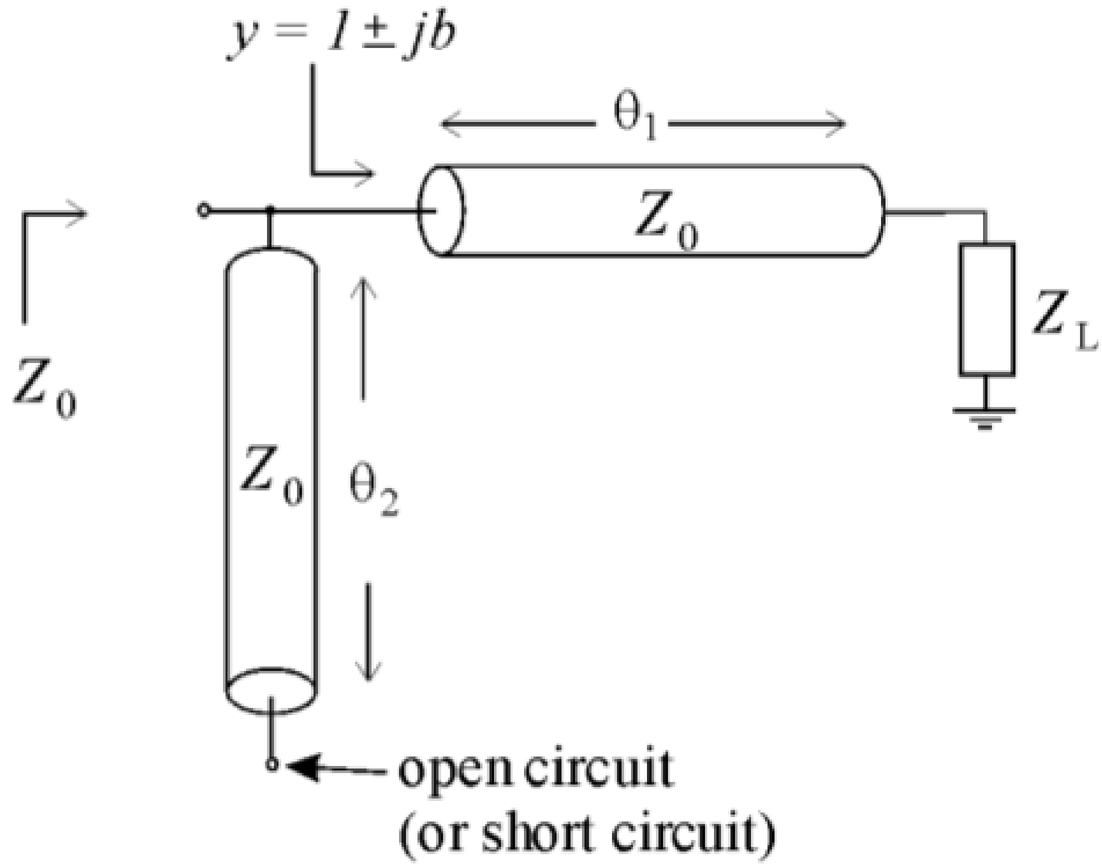
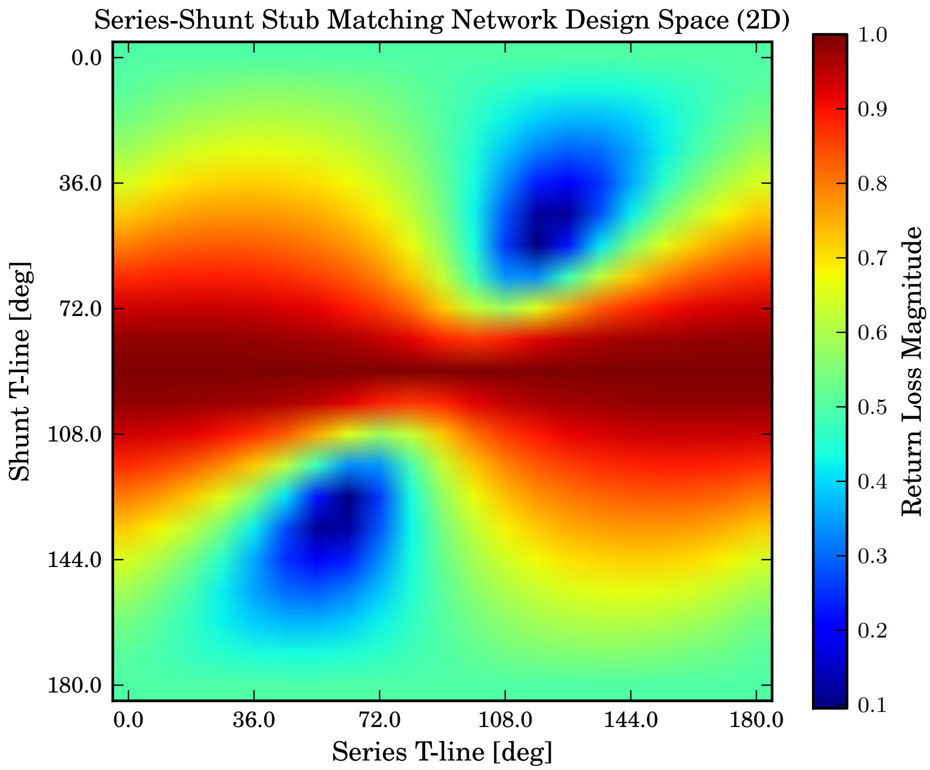
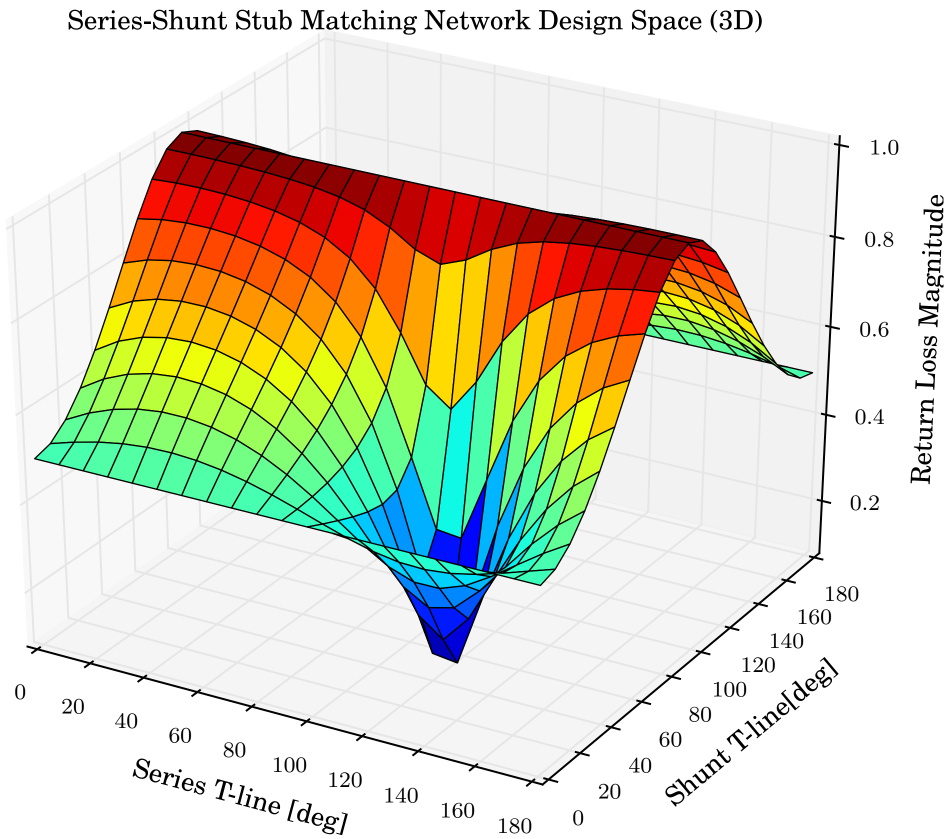


Figure 2.1: Single stub matching Network

```
    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 (2D)
0.0
36.0
72.0
108.0
144.0
180.0
Shunt T-line [deg]
0.0
36.0
72.0
108.0
144.0
180.0
Series T-line [deg]
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
Return Loss Magnitude



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.s1p'),
        rf.Network('ideal/open.s1p'),
        rf.Network('ideal/load.s1p'),
        ]

# a list of Network types, holding 'measured' responses
my_measured = [\
        rf.Network('measured/short.s1p'),
        rf.Network('measured/open.s1p'),
        rf.Network('measured/load.s1p'),
        ]
```

```
## 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.s1p')
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  may be similar to above example
```

CHAPTER

THREE

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 fruequency 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 frquency 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

center	Center frequency.
f	Frequency vector in Hz
f_scaled	Frequency vector in units of unit
multiplier	Multiplier for formating axis
multiplier_dict	
npoints	starting frequency in Hz
span	the frequency span
start	starting frequency in Hz
step	the inter-frequency step size

Continued on next page

Table 3.2 – continued from previous page

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

skrf.frequency.Frequency.center

Frequency.**center**

Center frequency.

Returns **center** : number

the 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 formating 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

`__init__`	Frequency initializer.
`copy`	returns a new copy of this frequency
`from_f`	Alternative constructor of a Frequency object from a frequency
`labelXAxis`	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** : number

start frequency in units of *unit*

stop : number

stop 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

```
>>> wr1p5band = 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.x_label()` with string containing the frequency unit.

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

Axes on which to label the plot, defaults what is returned by `matplotlib.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 [1].

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
s	scattering parameter matrix
z0	characteristic impedance matrix
f	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
s_re	real part of the s-matrix
s_im	imaginary part of the s-matrix
s_mag	magnitude of the s-matrix
s_db	magnitude in log scale of the s-matrix
s_deg	phase of the s-matrix in degrees

The following operations act on the networks s-matrix.

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

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.

[1] http://en.wikipedia.org/wiki/Two-port_network

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

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

Continued on next page

Table 3.5 – continued from previous page

t	Scattering transfer parameters
y	Admittance parameter matrix.
y_arcl	The arcl component of the y-matrix ..
y_arcl_unwrap	The arcl_unwrap component of the y-matrix ..
y_db	The db component of the y-matrix ..
y_deg	The deg component of the y-matrix ..
y_deg_unwrap	The deg_unwrap component of the y-matrix ..
y_im	The im component of the y-matrix ..
y_mag	The mag component of the y-matrix ..
y_rad	The rad component of the y-matrix ..
y_rad_unwrap	The rad_unwrap component of the y-matrix ..
y_re	The re component of the y-matrix ..
z	Impedance parameter matrix.
z0	Characteristic impedance[s] of the network ports.
z_arcl	The arcl component of the z-matrix ..
z_arcl_unwrap	The arcl_unwrap component of the z-matrix ..
z_db	The db component of the z-matrix ..
z_deg	The deg component of the z-matrix ..
z_deg_unwrap	The deg_unwrap component of the z-matrix ..
z_im	The im component of the z-matrix ..
z_mag	The mag component of the z-matrix ..
z_rad	The rad component of the z-matrix ..
z_rad_unwrap	The rad_unwrap component of the z-matrix ..
z_re	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 a11 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-embeding. 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 who's diagonals are equal to the total power received at all ports, normalized to the power at a single excitement port.

mathmatically, this is a test for unitary-ness of the s-parameter matrix [2].

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 fxnxn

[2] 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 *fxnxn*, 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 *fxnxn*

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 [3] is a 3-dimensional `numpy.ndarray` which has shape *fx2x2*, 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.ndarry of shape *fx2x2*

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 [4] 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 y11 can be accessed by taking the slice *y[:,0,0]*.

[3] http://en.wikipedia.org/wiki/Scattering_parameters#Scattering_transfer_parameters

[4] 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 [5] 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 z11 can be accessed by taking the slice *z[:,0,0]*.

Returns **z** : complex `numpy.ndarray` of shape *fxnxn*

[5] 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

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

Continued on next page

Table 3.6 – continued from previous page

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

multiplys 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. defualt 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

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_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

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(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

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_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

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_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

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_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

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_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

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(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

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_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

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_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

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_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

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(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

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_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

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(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

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_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

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_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

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(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

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_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

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_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

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_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 addmitance 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

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(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

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(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

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_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

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(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

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_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

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_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

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_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

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(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

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(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

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_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

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_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

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(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

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_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

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_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

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_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.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.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

`connect`(ntwkA, k, ntwkB, l[, num])	connect two n-port networks together.
`innerconnect`(ntwkA, k, l[, num])	connect ports of a single n-port network.
`cascade`(ntwkA, ntwkB)	Cascade two 2-port Networks together
`de_embed`(ntwkA, ntwkB)	De-embed *ntwkA* from *ntwkB*.
`flip`(a)	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 recomended 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-embeded 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

`Network.resample`(npoints, **kwargs)	Interpolate network based on a new number of frequency points
`Network.interpolate`(new_frequency, **kwargs)	Return an interpolated network, from a new :class:'~skrf.frequency
`Network.interpolate_self`(new_frequency, **kwargs)	Interpolates s-parameters given a new
`Network.interpolate_from_f`(f[, interp_kwargs])	Interpolates s-parameters from a frequency vector.
`stitch`(ntwkA, ntwkB, **kwargs)	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.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.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

`skrf.io.general.read`(file, *args, **kwargs)	Read skrf object[s] from a pickle file
`skrf.io.general.write`(file, obj[, overwrite])	Write skrf object[s] to a file
`Network.write`([file])	Write the Network to disk using the `pickle` module.
`Network.write_touchstone`([filename, dir, ...])	write a contents of the `Network` to a touchstone file.

Continued on next page

Table 3.9 – continued from previous page

Network.read(*args, **kwargs)	Read a Network from a 'ntwk' file

3.2.5 Noise

Network.add_noise_polar(mag_dev, phase_dev, ...)	adds a complex zero-mean gaussian white-noise.
Network.add_noise_polar_flatband(mag_dev, ...)	adds a flatband complex zero-mean gaussian white-noise signal of
Network.multiply_noise(mag_dev, phase_dev, ...)	multiplys 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*)

multiplys 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. defualt to 1

returns: nothing

3.2.6 Supporting Functions

inv(s)	Calculates 'inverse' s-parameter matrix, used for de-embeding
connect_s(A, k, B, l)	connect two n-port networks' s-matricies together.

Continued on next page

Table 3.11 – continued from previous page

innerconnect_s(A, k, l)	connect two ports of a single n-port network's s-matrix.
s2z(s[, z0])	Convert scattering parameters [#]_ to impedance parameters [#]_ ..
s2y(s[, z0])	convert scattering parameters [#]_ to admittance parameters [#]_
s2t(s)	Converts scattering parameters [#]_ to scattering transfer parameters [#]_ .
z2s(z[, z0])	convert impedance parameters [#]_ to scattering parameters [#]_
z2y(z)	convert impedance parameters [#]_ to admittance parameters [#]_
z2t(z)	Not Implemented yet
y2s(y[, z0])	convert admittance parameters [#]_ to scattering parameters [#]_
y2z(y)	convert admittance parameters [#]_ to impedance parameters [#]_
y2t(y)	Not Implemented Yet
t2s(t)	converts scattering transfer parameters [#]_ to scattering parameters [#]_
t2z(t)	Not Implemented Yet
t2y(t)	Not Implemented Yet

skrf.network.inv

skrf.network.**inv**(*s*)

Calculates 'inverse' s-parameter matrix, used for de-embeding

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-matricies 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-matricies. The function connect() operates on Network types.

Parameters **A** : numpy.ndarray

S-parameter matrix of *A*, shape is fxnxn

k : int

port index on *A* (port indices start from 0)

B : numpy.ndarray

S-parameter matrix of *B*, shape is fxnxn

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-matricies. The function innerconnect() operates on Network types.

Parameters **A** : numpy.ndarray

S-parameter matrix of *A*, shape is fxnxn

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 [6]. The original paper describing the algorithm is given in [7].

[6] 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

[7] 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 [8] to impedance parameters [9]

$$z = \sqrt{z_0} \cdot (I + s)(I - s)^{-1} \cdot \sqrt{z_0}$$

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, y2z, t2s, t2z, t2y, Network.s, Network.y, Network.z, Network.t

References

skrf.network.s2y

`skrf.network.s2y(s, z0=50)`

convert scattering parameters [10] to admittance parameters [11]

$$y = \sqrt{y_0} \cdot (I - s)(I + s)^{-1} \cdot \sqrt{y_0}$$

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, y2z, t2s, t2z, t2y, Network.s, Network.y, Network.z, Network.t

[8] http://en.wikipedia.org/wiki/S-parameters
[9] http://en.wikipedia.org/wiki/impedance_parameters
[10] http://en.wikipedia.org/wiki/S-parameters
[11] http://en.wikipedia.org/wiki/Admittance_parameters

References

skrf.network.s2t

skrf.network.**s2t**(*s*)

Converts scattering parameters [12] to scattering transfer parameters [13] .

transfer parameters are also refered 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 [14] to scattering parameters [15]

$$s = (\sqrt{y_0} \cdot z \cdot \sqrt{y_0} - I)(\sqrt{y_0} \cdot z \cdot \sqrt{y_0} + I)^{-1}$$

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

[12] http://en.wikipedia.org/wiki/S-parameters
[13] http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters
[14] http://en.wikipedia.org/wiki/impedance_parameters
[15] http://en.wikipedia.org/wiki/S-parameters

References

skrf.network.z2y

skrf.network.**z2y**(*z*)

convert impedance parameters [16] to admittance parameters [17]

$$y = z^{-1}$$

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 [18] to scattering transfer parameters [19]

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`

[16] http://en.wikipedia.org/wiki/impedance_parameters
[17] http://en.wikipedia.org/wiki/Admittance_parameters
[18] http://en.wikipedia.org/wiki/impedance_parameters
[19] http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters

References

skrf.network.y2s

`skrf.network.y2s(y, z0=50)`

convert admittance parameters [20] to scattering parameters [21]

$$s = (I - \sqrt{z_0} \cdot y \cdot \sqrt{z_0})(I + \sqrt{z_0} \cdot y \cdot \sqrt{z_0})^{-1}$$

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 [22] to impedance parameters [23]

$$z = y^{-1}$$

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`

[20] http://en.wikipedia.org/wiki/Admittance_parameters
[21] http://en.wikipedia.org/wiki/S-parameters
[22] http://en.wikipedia.org/wiki/Admittance_parameters
[23] http://en.wikipedia.org/wiki/impedance_parameters

References

skrf.network.y2t

`skrf.network.y2t(y)`

Not Implemented Yet

convert admittance parameters [24] to scattering-transfer parameters [25]

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 [26] to scattering parameters [27]

transfer parameters are also refered 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

[24] http://en.wikipedia.org/wiki/Admittance_parameters
[25] http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters
[26] http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters
[27] http://en.wikipedia.org/wiki/S-parameters

Convert scattering transfer parameters [28] to impedance parameters [29]

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 [30]

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

`average`(list_of_networks)	Calculates the average network from a list of Networks.
`Network.nudge`([amount])	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`

[28] http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters
[29] http://en.wikipedia.org/wiki/impedance_parameters
[30] 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.s1p'), rf.Network('myntwk2.s1p')]
>>> 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 ititialization, 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 accessable 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

`inv`	
`mean_s_db`	the mean magnitude in dB.
`std_s_db`	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 convertedto 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 convertedto db, so magnitude_2_db(mean(s_mag))

which is NOT the same as mean(s_db)

Methods

`__init__`	Initializer for NetworkSet
`copy`	copies each network of the network set.
`element_wise_method`	calls a given method of each element and returns the result as
`from_zip`	creates a NetworkSet from a zipfile of touchstones.
`plot_logsigma`	plots the uncertainty for the set in units of log-sigma.
`plot_uncertainty_bounds_component`	plots mean value of the NetworkSet with +- uncertainty bounds
`plot_uncertainty_bounds_s`	Plots complex uncertianty bounds plot on smith chart.
`plot_uncertainty_bounds_s_db`	this just calls
`plot_uncertainty_decomposition`	plots the total and component-wise uncertainty
`set_wise_function`	calls a function on a specific property of the networks in
`signature`	visualization of relative changes in a NetworkSet.
`uncertainty_ntwk_triplet`	returns a 3-tuple of Network objects which contain the
`write`	Write the NetworkSet to disk using `write()`

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 ploting the uncertainty in the magnitude would be expressed by,

mean(abs(s)) +- std(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. uncerainty for wrapped phase blows up at +-pi.

skrf.networkSet.NetworkSet.plot_uncertainty_bounds_s

NetworkSet.**plot_uncertainty_bounds_s**(*multiplier=200*, **args*, ***kwargs*)

Plots complex uncertianty 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 **multipliter** : 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 devation 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

smith([smithR, chart_type, draw_labels, ...])	plots the smith chart of a given radius
plot_smith(z[, smith_r, chart_type, ...])	plot complex data on smith chart
plot_rectangular(x, y[, x_label, y_label, ...])	plots rectangular data and optionally label axes.
plot_polar(theta, r[, x_label, y_label, ...])	plots polar data on a polar plot and optionally label axes.
plot_complex_rectangular(z[, x_label, ...])	plot complex data on the complex plane
plot_complex_polar(z[, x_label, y_label, ...])	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

`save_all_figs`([dir, format])	Save all open Figures to disk.
`add_markers_to_lines`([ax, marker_list, ...])	adds markers to existing lings on a plot
`legend_off`([ax])	turn off the legend for a given axes.
`func_on_all_figs`(func, *args, **kwargs)	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 lings on a plot

this is convinient 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** : pased 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

complex_2_reim(z)	takes:
complex_2_magnitude(input)	returns the magnitude of a complex number.
complex_2_db(input)	returns the magnitude in dB of a complex number.
complex_2_radian(input)	returns the angle complex number in radians.
complex_2_degree(input)	returns the angle complex number in radians.
complex_2_magnitude(input)	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*log10(|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

unwrap_rad(input)	unwraps a phase given in radians
sqrt_phase_unwrap(input)	takes the square root of a complex number with unwraped 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 unwraped phase

this idea came from Lihan Chen

3.5.3 Unit Conversion

radian_2_degree(rad)	
degree_2_radian(deg)	
np_2_db(x)	converts a value in dB to neper's
db_2_np(x)	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.

complex2Scalar(input)	
scalar2Complex(input)	

skrf.mathFunctions.complex2Scalar

`skrf.mathFunctions.complex2Scalar(input)`

skrf.mathFunctions.scalar2Complex

`skrf.mathFunctions.scalar2Complex(input)`

3.5.5 Special Functions

dirac_delta(x)	the dirac function.
neuman(x)	neumans number
null(A[, eps])	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 tranmission 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	z_l	load_impedance
Z_{in}	z_in	input_impedance
Γ_0	Gamma_0	reflection_coefficient
Γ_{in}	Gamma_in	reflection_coefficient_at_theta
θ	theta	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 θ. The line is terminated on the right with some load impedance, Z_l. The input impedance Z_{in} and input reflection coefficient Γ_{in} are looking in towards the load from the distance θ from the load.

$$Z_0, \theta$$

o===============o=$[Z_l]$

$$\rightarrow \qquad\qquad\qquad\qquad \rightarrow$$

$$Z_{in}, \Gamma_{in} \qquad\qquad\qquad\qquad Z_l, \Gamma_0$$

So, to clarify the confusion,

$$Z_{in} = Z_l, \qquad \Gamma_{in} = \Gamma_l \text{ at } \theta = 0$$

Short names

`theta`(gamma, f, d[, deg])	Calculates the electrical length of a section of transmission line.
`zl_2_Gamma0`(z0, zl)	Returns the reflection coefficient for a given load impedance, and characteristic impedan
`Gamma0_2_zl`(z0, Gamma)	calculates the input impedance given a reflection coefficient and
`zl_2_zin`(z0, zl, theta)	input impedance of load impedance zl at a given electrical length,
`zl_2_Gamma_in`(z0, zl, theta)	
`Gamma0_2_Gamma_in`(Gamma0, theta)	reflection coefficient at a given electrical length.
`Gamma0_2_zin`(z0, Gamma0, theta)	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_zl` reflection coefficient to load impedance

Notes

inputs are typecasted to 1D complex array

skrf.tlineFunctions.Gamma0_2_zl

`skrf.tlineFunctions.`**`Gamma0_2_zl`**(*z0*, *Gamma*)

calculates the input impedance given a reflection coefficient and characterisitc impedance

$$Z_0 (\frac{1+\Gamma}{1-\Gamma})$$

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`**(*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.Gamma0_2_zin

skrf.tlineFunctions.**Gamma0_2_zin**(*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

Long-names

distance_2_electrical_length(gamma, f, d[, deg])	Calculates the electrical length of a section of trans
electrical_length_2_distance(theta, gamma, f0)	Convert electrical length to a physical distance.
reflection_coefficient_at_theta(Gamma0, theta)	reflection coefficient at a given electrical length.
reflection_coefficient_2_input_impedance(z0, ...)	calculates the input impedance given a reflection c
reflection_coefficient_2_input_impedance_at_theta(z0, ...)	calculates the input impedance at electrical length
input_impedance_at_theta(z0, zl, theta)	input impedance of load impedance zl at a given el
load_impedance_2_reflection_coefficient(z0, zl)	Returns the reflection coefficient for a given load i
load_impedance_2_reflection_coefficient_at_theta(z0, ...)	

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

$$Z_0(\frac{1+\Gamma}{1-\Gamma})$$

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_zl` 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

`distributed_circuit_2_propagation_impedance`(...)	Converts distrubuted circuit values to wave quantities.
`propagation_impedance_2_distributed_circuit`(...)	Converts wave quantities to distrubuted circuit values.

skrf.tlineFunctions.distributed_circuit_2_propagation_impedance

skrf.tlineFunctions.**distributed_circuit_2_propagation_impedance**(*distributed_admittance*, *distributed_impedance*)

Converts distrubuted circuit values to wave quantities.

This converts complex distributed impedance and admittance to propagation constant and characteristic impedance. The relation is

$$Z_0 = \sqrt{\frac{Z'}{Y'}} \qquad \gamma = \sqrt{Z'Y'}$$

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 distrubuted circuit values.

Converts complex propagation constant and characteristic impedance to distributed impedance and admittance. The relation is,

$$Z^{'} = \gamma Z_0 \qquad Y^{'} = \frac{\gamma}{Z_0}$$

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

`skin_depth`(f, rho, mu_r)	the skin depth for a material.
`surface_resistivity`(f, rho, mu_r)	surface resistivity.

skrf.tlineFunctions.skin_depth

`skrf.tlineFunctions.skin_depth`(*f*, *rho*, *mu_r*)
the skin depth for a material.

see www.microwaves101.com for more info.

Parameters **f** : number or array-like

frequency, in Hz

rho : number of array-like

bulk resistivity of material, in ohm*m

mu_r : number or array-like

relative permiability 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 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 permiability 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 [31] .

Object Name	Description
f_wr10	WR-10, 75-110 GHz
f_wr3	WR-3, 220-325 GHz
f_wr2p2	WR-2.2, 330-500 GHz
f_wr1p5	WR-1.5, 500-750 GHz
f_wr1	WR-1, 750-1100 GHz
...	...

RectangularWaveguide Objects

These are predefined RectangularWaveguide objects for standard waveguide bands.

Object Name	Description
wr10	WR-10, 75-110 GHz
wr3	WR-3, 220-325 GHz
wr2p2	WR-2.2, 330-500 GHz
wr1p5	WR-1.5, 500-750 GHz
wr1	WR-1, 750-1100 GHz
...	...

3.7.2 Shorthand Names

Below is a list of shorthand object names which can be use to save some typing. These names are defined in the main __*init*__ module.

Shorthand	Full Object Name
F	Frequency
N	Network
NS	NetworkSet
M	Media
C	Calibration

The following are shorthand names for commonly used, but unfortunately longwinded functions.

Shorthand	Full Object Name
saf	save_all_figs()

3.7.3 References

3.8 util (skrf.util)

Holds utility functions that are general conveniences.

[31] VDI Application Note: VDI Waveguide Band Designations (VDI-1002) http://vadiodes.com/VDI/pdf/waveguidechart200908.pdf

3.8.1 General

now_string()	returns a unique sortable string, representing the current time
find_nearest(array, value)	find nearest value in array.
find_nearest_index(array, value)	find nearest value in array.
get_fid(file, *args, **kwargs)	Returns a file object, given a filename or file object
get_extn(filename)	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 :

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

`read`(file, *args, **kwargs)	Read skrf object[s] from a pickle file
`read_all`([dir, contains])	Read all skrf objects in a directory
`write`(file, obj[, overwrite])	Write skrf object[s] to a file
`write_all`(dict_objs[, dir])	Write a dictionary of skrf objects individual files in *dir*.
`save_sesh`(dict_objs[, file, module, ...])	Save all *skrf* 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 :

dont 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 [32]

Methods

[32] http://www.eda-stds.org/ibis/adhoc/interconnect/touchstone_spec2_draft.pdf

`__init__`	constructor
`get_format`	returns the file format string used for the given format.
`get_noise_data`	TODO: NIY
`get_noise_names`	TODO: NIY
`get_sparameter_arrays`	returns the sparameters as a tuple of arrays, where the first element is
`get_sparameter_data`	get the data of the sparameter with the given format.
`get_sparameter_names`	generate a list of column names for the s-parameter data
`load_file`	Load the touchstone file into the interal 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_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 magnitute 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. posible 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 interal data structures

Functions related to reading/writing touchstones.

hfss_touchstone_2_gamma_z0(filename)	Extracts Z0 and Gamma comments from touchstone file
hfss_touchstone_2_media(filename[, f_unit])	Creates a Media object from a a HFSS-style touchstone file with Gamma and

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 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')
```

3.10 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 meaurements and ideals responses. It can run a calibration, store results, and apply the results to calculate corrected measurements.

Attributes

`Ts`	T-matricies used for de-embeding, a two-port calibration.
`caled_ntwk_sets`	returns a NetworkSet for each caled_ntwk, based on their names
`caled_ntwks`	list of the calibrated, calibration standards.
`calibration_algorithm_dict`	
`coefs`	coefs: a dictionary holding the calibration coefficients
`error_ntwk`	A Network object which represents the error network being
`nports`	the number of ports in the calibration
`nstandards`	number of ideal/measurement pairs in calibration
`output_from_cal`	a dictionary holding all of the output from the calibration
`residual_ntwks`	returns a the residuals for each calibration standard in the
`residuals`	if calibration is overdeteremined, this holds the residuals
`type`	string representing what type of calibration is to be

skrf.calibration.calibration.Calibration.Ts

`Calibration.`**`Ts`**

T-matricies used for de-embeding, 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 0x471dc80>, '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

`__init__`	Calibration initializer.
`apply_cal`	apply the current calibration to a measurement.

Continued on next page

Table 3.34 – continued from previous page

apply_cal_to_all_in_dir	convience function to apply calibration to an entire directory
biased_error	estimate of biased error for overdetermined calibration with
func_per_standard	
mean_residuals	
plot_coefs_db	plot magnitude of the error coeficient dictionary
plot_errors	plot calibration error metrics for an over-determined calibration.
plot_residuals	plots a component of the residual errors on the Calibration-plane.
plot_residuals_db	see plot_residuals
plot_residuals_mag	see plot_residuals
plot_residuals_smith	see plot_residuals
plot_uncertainty_per_standard	Plots uncertainty associated with each calibration standard.
run	runs the calibration algorihtm.
total_error	estimate of total error for overdetermined calibration with
unbiased_error	estimate of unbiased error for overdetermined calibration with
uncertainty_per_standard	given that you have repeat-connections of single standard,
write	Write the Calibration to disk using `write()`

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_algorihtm_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')`

convience 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 mean_s(**|mean_c(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 ploting method of Network class to call

possible options are: 'mag', 'db', 'smith', 'deg', etc

*args,**kwargs: passed to **plot_s_**'atttribute'()

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 algorihtm.

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 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 mean_s(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

`one_port`(measured, ideals)	Standard algorithm for a one port calibration.
`one_port_nls`(measured, ideals)	one port non-linear least squares.
`two_port`(measured, ideals[, switch_terms])	Two port calibration based on the 8-term error model.
`parameterized_self_calibration`(measured, ideals)	An iterative, general self-calibration routine.
`parameterized_self_calibration_nls`(measured, ...)	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 coeffcients': 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 coeffcients': 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.optmize.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 [33] can be taken into account by passing a tuple containing the forward and reverse switch terms as 1-port Networks. This algorithm is based on the work in [34] .

[33] 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

[34] 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 [3] 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_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

unterminate_switch_terms(two_port, gamma_f, ...)	unterminates switch terms from raw measurements.
abc_2_coefs_dict(abc)	converts an abc ndarry to a dictionarry containing the error
eight_term_2_one_port_coefs(coefs)	

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 ndarry to a dictionarry 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) = e01*e10 - e00*e11 b = e00 c = e11

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

cartesian_product_calibration_set(ideals, ...)	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*, *measured*, **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 meaurements and ideals responses. It can run a calibration, store results, and apply the results to calculate corrected measurements.

Attributes

`Ts`	T-matricies used for de-embeding, a two-port calibration.
`caled_ntwk_sets`	returns a NetworkSet for each caled_ntwk, based on their names
`caled_ntwks`	list of the calibrated, calibration standards.
`calibration_algorithm_dict`	
`coefs`	coefs: a dictionary holding the calibration coefficients
`error_ntwk`	A Network object which represents the error network being
`nports`	the number of ports in the calibration
`nstandards`	number of ideal/measurement pairs in calibration
`output_from_cal`	a dictionary holding all of the output from the calibration
`residual_ntwks`	returns a the residuals for each calibration standard in the
`residuals`	if calibration is overdeteremined, this holds the residuals
`type`	string representing what type of calibration is to be

skrf.calibration.calibration.Calibration.Ts

Calibration.**Ts**
T-matricies used for de-embeding, 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 0x471dc80>, '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

`__init__`	Calibration initializer.
`apply_cal`	apply the current calibration to a measurement.
`apply_cal_to_all_in_dir`	convience function to apply calibration to an entire directory
`biased_error`	estimate of biased error for overdetermined calibration with
`func_per_standard`	
`mean_residuals`	
`plot_coefs_db`	plot magnitude of the error coeficient dictionary
`plot_errors`	plot calibration error metrics for an over-determined calibration.
`plot_residuals`	plots a component of the residual errors on the Calibration-plane.
`plot_residuals_db`	see plot_residuals
`plot_residuals_mag`	see plot_residuals
`plot_residuals_smith`	see plot_residuals

Continued on next page

Table 3.40 – continued from previous page

plot_uncertainty_per_standard	Plots uncertainty associated with each calibration standard.
run	runs the calibration algorihtm.
total_error	estimate of total error for overdetermined calibration with
unbiased_error	estimate of unbiased error for overdetermined calibration with
uncertainty_per_standard	given that you have repeat-connections of single standard,
write	Write the Calibration to disk using write()

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_algorihtm_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'*)

convience 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 mean_s(**|mean_c(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 ploting method of Network class to call

possible options are: 'mag', 'db', 'smith', 'deg', etc

*args,**kwargs: passed to **plot_s_**'atttribute'()

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 algorihtm.

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 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 mean_s(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 transmision line medium, such as `line()` and `delay_short()`. These methods are inherited by the specific tranmission line classes, which interally 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

`Media`	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 transmision 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 specicial load cases, suc 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

`characteristic_impedance`	Characterisitc impedance
`propagation_constant`	Propagation constant
`z0`	Port Impedance

skrf.media.media.Media.characteristic_impedance

`Media.`**`characteristic_impedance`**

Characterisitc 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 characterisitc 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 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.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 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

`__init__`	The Media initializer.
`capacitor`	Capacitor
`delay_load`	Delayed load
`delay_open`	Delayed open transmission line
`delay_short`	Delayed Short
`electrical_length`	calculates the electrical length for a given distance, at
`from_csv`	create a Media from numerical values stored in a csv file.
`guess_length_of_delay_short`	Guess physical length of a delay short.
`impedance_mismatch`	Two-port network for an impedance miss-match
`inductor`	Inductor
`line`	Matched transmission line of given length
`load`	Load of given reflection coefficient.
`match`	Perfect matched load ($\Gamma_0 = 0$).
`open`	Open ($\Gamma_0 = 1$)
`resistor`	Resistor
`short`	Short ($\Gamma_0 = -1$)
`shunt`	Shunts a `Network`
`shunt_capacitor`	Shunted capacitor
`shunt_delay_load`	Shunted delayed load
`shunt_delay_open`	Shunted delayed open
`shunt_delay_short`	Shunted delayed short
`shunt_inductor`	Shunted inductor
`splitter`	Ideal, lossless n-way splitter.
`tee`	Ideal, lossless tee.

Continued on next page

Table 3.43 – continued from previous page

theta_2_d	Converts electrical length to physical distance.
thru	Matched transmission line of length 0.
white_gaussian_polar	Complex zero-mean gaussian white-noise network.
write_csv	write this media's frequency, z0, 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*, *characterisitc_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 characterisitc impedance of the transmission line medium (None) [a number]. if z0= None then will set to characterisitc_impedance

See Also:

from_csv() function to create a Media object from a csv file containing gamma/z0

Notes

propagation_constant **must adhere to the following convention,**

- positive real(gamma) = attenuation
- positive imag(gamma) = forward propagation

the z0 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 z0=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 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.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 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.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 electral 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, 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 measurment 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 **missmatch** : `Network` object

a 2-port network representing the impedance missmatch

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 tranmission 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: kxnxn, 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 = Gamma0*eye(...)

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

characterisitc 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 + 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 capcitor(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 tranmission 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

DistributedCircuit	Generic, distributed circuit TEM transmission line
RectangularWaveguide	Rectangular Waveguide medium.
CPW	Coplanar waveguide class
Freespace	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 (ω):

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	$Z_0 = \sqrt{\frac{Z'}{Y'}}$	Z0()
Propagation Constant	$\gamma = \sqrt{Z'Y'}$	gamma()

Given the following definitions, the components of propagation constant are interpreted as follows:

$$+\Re e\{\gamma\} = \text{attenuation}$$
$$-\Im m\{\gamma\} = \text{forward propagation}$$

Attributes

`Y`	Distributed Admittance, $Y^{'}$
`Z`	Distributed Impedance, $Z^{'}$
`characteristic_impedance`	Characterisitc impedance
`propagation_constant`	Propagation constant
`z0`	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

$$Z^{'} = R^{'} + j\omega I^{'}$$

Returns **Z** : numpy.ndarray

Distributed impedance in units of ohm/m

skrf.media.distributedCircuit.DistributedCircuit.characteristic_impedance

`DistributedCircuit.characteristic_impedance`
Characterisitc 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 characterisitc 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 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

`Z0`	Characteristic Impedance, $Z0$
`__init__`	Distributed Circuit constructor.
`capacitor`	Capacitor
`delay_load`	Delayed load
`delay_open`	Delayed open transmission line
`delay_short`	Delayed Short
`electrical_length`	calculates the electrical length for a given distance, at
`from_Media`	Initializes a DistributedCircuit from an existing
`from_csv`	create a Media from numerical values stored in a csv file.
`gamma`	Propagation Constant, γ
`guess_length_of_delay_short`	Guess physical length of a delay short.
`impedance_mismatch`	Two-port network for an impedance miss-match
`inductor`	Inductor
`line`	Matched transmission line of given length
`load`	Load of given reflection coefficient.
`match`	Perfect matched load ($\Gamma_0 = 0$).
`open`	Open ($\Gamma_0 = 1$)
`resistor`	Resistor
`short`	Short ($\Gamma_0 = -1$)
`shunt`	Shunts a `Network`

Continued on next page

Table 3.46 – continued from previous page

shunt_capacitor	Shunted capacitor
shunt_delay_load	Shunted delayed load
shunt_delay_open	Shunted delayed open
shunt_delay_short	Shunted delayed short
shunt_inductor	Shunted inductor
splitter	Ideal, lossless n-way splitter.
tee	Ideal, lossless tee.
theta_2_d	Converts electrical length to physical distance.
thru	Matched transmission line of length 0.
white_gaussian_polar	Complex zero-mean gaussian white-noise network.
write_csv	write this media's frequency, z0, and gamma to a csv file.

skrf.media.distributedCircuit.DistributedCircuit.Z0

`DistributedCircuit.`**`Z0`**`()`

Characteristic Impedance, $Z0$

$$Z_0 = \sqrt{\frac{Z'}{Y'}}$$

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_open** : `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, γ

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 measurment 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 missmatch

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: kxnxn, 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 = Gamma0*eye(...)

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

characterisitc 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 + 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 capcitor(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 tranmission 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 indecies (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

characteristic_impedance	Characterisitc impedance
ep	The permativity of the filling material
k0	Characteristic wave number
kc	Cut-off wave number
kx	Eigen value in the 'a' direction
ky	Eigen-value in the *b* direction.
mu	The permeability of the filling material
propagation_constant	Propagation constant
z0	Port Impedance

skrf.media.rectangularWaveguide.RectangularWaveguide.characteristic_impedance

`RectangularWaveguide.characteristic_impedance`

Characterisitc 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 characterisitc 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

$$k_c = \sqrt {k_x^2 + k_y^2} = \sqrt {m \frac{\pi}{a}^2 + n \frac{\pi}{b}^2}$$

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 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.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 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

Z0	The characteristic impedance
__init__	RectangularWaveguide initializer
capacitor	Capacitor
delay_load	Delayed load
delay_open	Delayed open transmission line
delay_short	Delayed Short
electrical_length	calculates the electrical length for a given distance, at
from_csv	create a Media from numerical values stored in a csv file.
guess_length_of_delay_short	Guess physical length of a delay short.
impedance_mismatch	Two-port network for an impedance miss-match
inductor	Inductor
kz	The Longitudinal wave number, aka propagation constant.
line	Matched transmission line of given length
load	Load of given reflection coefficient.
match	Perfect matched load ($\Gamma_0 = 0$).
open	Open ($\Gamma_0 = 1$)
resistor	Resistor
short	Short ($\Gamma_0 = -1$)
shunt	Shunts a Network
shunt_capacitor	Shunted capacitor
shunt_delay_load	Shunted delayed load
shunt_delay_open	Shunted delayed open
shunt_delay_short	Shunted delayed short
shunt_inductor	Shunted inductor
splitter	Ideal, lossless n-way splitter.
tee	Ideal, lossless tee.
theta_2_d	Converts electrical length to physical distance.
thru	Matched transmission line of length 0.
white_gaussian_polar	Complex zero-mean gaussian white-noise network.
write_csv	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, keywrod 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 *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.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, 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 measurment 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 missmatch

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 tranmission 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: kxnxn, 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 = Gamma0*eye(...)

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

characterisitc 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 + 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 tranmission 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 [35] provided by the qucs project [36] . The variables and properties of this class are coincident with their derivations.

Attributes

`K_ratio`	intermediary parameter. see qucs docs on cpw lines.
`alpha_conductor`	Losses due to conductor resistivity
`characteristic_impedance`	Characterisitc impedance
`ep_re`	intermediary parameter. see qucs docs on cpw lines.
`k1`	intermediary parameter. see qucs docs on cpw lines.
`propagation_constant`	Propagation constant
`z0`	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

[35] http://qucs.sourceforge.net/docs/technical.pdf
[36] http://www.qucs.sourceforge.net/

skrf.media.cpw.CPW.characteristic_impedance

`CPW.characteristic_impedance`

Characterisitc 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 characterisitc 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

`Z0`	Characterisitc impedance
`__init__`	Coplanar Waveguide initializer
`capacitor`	Capacitor
`delay_load`	Delayed load
`delay_open`	Delayed open transmission line
`delay_short`	Delayed Short
`electrical_length`	calculates the electrical length for a given distance, at
`from_csv`	create a Media from numerical values stored in a csv file.
`gamma`	Propagation constant ..
`guess_length_of_delay_short`	Guess physical length of a delay short.
`impedance_mismatch`	Two-port network for an impedance miss-match
`inductor`	Inductor
`line`	Matched transmission line of given length
`load`	Load of given reflection coefficient.
`match`	Perfect matched load ($\Gamma_0 = 0$).
`open`	Open ($\Gamma_0 = 1$)
`resistor`	Resistor
`short`	Short ($\Gamma_0 = -1$)
`shunt`	Shunts a `Network`
`shunt_capacitor`	Shunted capacitor
`shunt_delay_load`	Shunted delayed load
`shunt_delay_open`	Shunted delayed open
`shunt_delay_short`	Shunted delayed short
`shunt_inductor`	Shunted inductor
`splitter`	Ideal, lossless n-way splitter.
`tee`	Ideal, lossless tee.
`theta_2_d`	Converts electrical length to physical distance.
`thru`	Matched transmission line of length 0.
`white_gaussian_polar`	Complex zero-mean gaussian white-noise network.
`write_csv`	write this media's frequency, z0, and gamma to a csv file.

skrf.media.cpw.CPW.Z0

`CPW.Z0()`

Characterisitc 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 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.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 measurment 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 missmatch

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: kxnxn, 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 = Gamma0*eye(...)

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

characterisitc 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 capcitor(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 tranmission 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 disctributed circuit transmission line model given in circuit theory by:

Circuit Property	Field Property
distributed_capacitance	real(ep_0*ep_r)
distributed_resistance	imag(ep_0*ep_r)
distributed_inductance	real(mu_0*mu_r)
distributed_conductance	imag(mu_0*mu_r)

This class's inheritence is; Media-> DistributedCircuit-> Freespace

Attributes

`Y`	Distributed Admittance, $Y^{'}$
`Z`	Distributed Impedance, $Z^{'}$
`characteristic_impedance`	Characterisitc impedance
`propagation_constant`	Propagation constant
`z0`	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`
Characterisitc 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 characterisitc 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 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.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 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

`Z0`	Characteristic Impedance, $Z0$
`__init__`	Freespace initializer
`capacitor`	Capacitor
`delay_load`	Delayed load
`delay_open`	Delayed open transmission line
`delay_short`	Delayed Short
`electrical_length`	calculates the electrical length for a given distance, at
`from_Media`	Initializes a DistributedCircuit from an existing
`from_csv`	create a Media from numerical values stored in a csv file.
`gamma`	Propagation Constant, γ
`guess_length_of_delay_short`	Guess physical length of a delay short.
`impedance_mismatch`	Two-port network for an impedance miss-match
`inductor`	Inductor
`line`	Matched transmission line of given length
`load`	Load of given reflection coefficient.
`match`	Perfect matched load ($\Gamma_0 = 0$).

Continued on next page

Table 3.52 – continued from previous page

open	Open ($\Gamma_0 = 1$)
resistor	Resistor
short	Short ($\Gamma_0 = -1$)
shunt	Shunts a Network
shunt_capacitor	Shunted capacitor
shunt_delay_load	Shunted delayed load
shunt_delay_open	Shunted delayed open
shunt_delay_short	Shunted delayed short
shunt_inductor	Shunted inductor
splitter	Ideal, lossless n-way splitter.
tee	Ideal, lossless tee.
theta_2_d	Converts electrical length to physical distance.
thru	Matched transmission line of length 0.
white_gaussian_polar	Complex zero-mean gaussian white-noise network.
write_csv	write this media's frequency, z0, and gamma 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(ep_0*ep_r)
distributed_resistance	imag(ep_0*ep_r)
distributed_inductance	real(mu_0*mu_r)
distributed_conductance	imag(mu_0*mu_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, γ

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 measurment 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 missmatch

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: kxnxn, 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 = Gamma0*eye(...)

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

characterisitc 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 + 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 capcitor(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 tranmission 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 vi (`skrf.vi`)

This module holds Virtual Instruments that are intricately related to **skrf**.

3.12.1 Vector Network Analyzers (`skrf.vi.vna`)

Warning: These Virtual Instruments are very spotily written, and may be subject to major re-writing in the future.

`PNAX`([address, channel])	Agilent PNAX
`ZVA40`([address, active_channel, continuous])	Rohde&Scharz ZVA40
`HP8510C`([address])	good ole 8510
`HP8720`([address])	

skrf.vi.vna.PNAX

class `skrf.vi.vna.`**`PNAX`**(*address=16, channel=1, **kwargs*)

Agilent PNAX

Examples

```
>>> from skrf.vi.vna import PNAX
>>> v = PNAX()
>>> dut = v.network
```

Attributes

continuous	
frequency	Gets frequency data, returning a Frequency object
network	Initiates a sweep and returns a Network type represting the data.

skrf.vi.vna.PNAX.continuous

PNAX.**continuous**

skrf.vi.vna.PNAX.frequency

PNAX.**frequency**

Gets frequency data, returning a Frequency object

Gets the

skrf.vi.vna.PNAX.network

PNAX.**network**

Initiates a sweep and returns a Network type represting the data.

If you are taking multiple sweeps, and want the sweep timing to work, put the turn continuous mode off, with pnax.continuous='off'

Examples

```
>>> from skrf.vi.vna import PNAX
>>> v = PNAX()
>>> dut = v.network
```

Methods

__init__

skrf.vi.vna.PNAX.__init__

PNAX.**__init__**(*address=16*, *channel=1*, ***kwargs*)

skrf.vi.vna.ZVA40

class skrf.vi.vna.**ZVA40**(*address=20*, *active_channel=1*, *continuous=True*, ***kwargs*)

Rohde&Scharz ZVA40

Examples

```
>>> from skrf.vi.vna import ZVA40
>>> v = ZVA40()
>>> dut = v.network
```

Attributes

continuous	set/get continuous sweep mode on/off [boolean]
error	returns list errors stored on vna
fdata	formated s-parameter data [a numpy array]
frequency	a frequency object, representing the current frequency axis
one_port	a network representing the current active trace
s11	this is just for legacy support, there is no gurantee this
sdata	unformated s-parameter data [a numpy array]

skrf.vi.vna.ZVA40.continuous

ZVA40.**continuous**
set/get continuous sweep mode on/off [boolean]

skrf.vi.vna.ZVA40.error

ZVA40.**error**
returns list errors stored on vna

skrf.vi.vna.ZVA40.fdata

ZVA40.**fdata**
formated s-parameter data [a numpy array]

skrf.vi.vna.ZVA40.frequency

ZVA40.**frequency**
a frequency object, representing the current frequency axis [skrf Frequency object]

skrf.vi.vna.ZVA40.one_port

ZVA40.**one_port**
a network representing the current active trace [skrf Network object]

skrf.vi.vna.ZVA40.s11

ZVA40.**s11**
this is just for legacy support, there is no gurantee this will return s11. it just returns active trace

skrf.vi.vna.ZVA40.sdata

ZVA40.**sdata**
unformated s-parameter data [a numpy array]

Methods

`__init__`	
`add_trace`	
`initiate`	initiate a sweep on current channel (low level timing)
`set_active_trace`	
`sweep`	initiate a sweep on current channel. if vna is in continous
`update_trace_list`	
`upload_cal_data`	for explanation of this code see the
`wait`	wait for preceding command to finish before executing subsequent

skrf.vi.vna.ZVA40.__init__

ZVA40.**__init__**(*address=20*, *active_channel=1*, *continuous=True*, ***kwargs*)

skrf.vi.vna.ZVA40.add_trace

ZVA40.**add_trace**(*parameter*, *name*)

skrf.vi.vna.ZVA40.initiate

ZVA40.**initiate**()
initiate a sweep on current channel (low level timing)

skrf.vi.vna.ZVA40.set_active_trace

ZVA40.**set_active_trace**(*name*)

skrf.vi.vna.ZVA40.sweep

ZVA40.**sweep**()
initiate a sweep on current channel. if vna is in continous mode it will put in single sweep mode, then request a sweep, and then return sweep mode to continous.

skrf.vi.vna.ZVA40.update_trace_list

ZVA40.**update_trace_list**()

skrf.vi.vna.ZVA40.upload_cal_data

ZVA40.**upload_cal_data**(*error_data*, *cal_name='test'*, *port=1*)
for explanation of this code see the zva manual (v1145.1084.12 p6.193)

skrf.vi.vna.ZVA40.wait

ZVA40.**wait**()
wait for preceding command to finish before executing subsequent commands

skrf.vi.vna.HP8510C

class skrf.vi.vna.**HP8510C**(*address=16*, ***kwargs*)
good ole 8510

Attributes

averaging	averaging factor
continuous	
error	
frequency	
one_port	Initiates a sweep and returns a Network type represting the
s11	
s12	
s21	
s22	
switch_terms	measures forward and reverse switch terms and returns them as a
two_port	Initiates a sweep and returns a Network type represting the

skrf.vi.vna.HP8510C.averaging

HP8510C.**averaging**
averaging factor

skrf.vi.vna.HP8510C.continuous

HP8510C.**continuous**

skrf.vi.vna.HP8510C.error

HP8510C.**error**

skrf.vi.vna.HP8510C.frequency

HP8510C.**frequency**

skrf.vi.vna.HP8510C.one_port

HP8510C.**one_port**

Initiates a sweep and returns a Network type represting the data.

if you are taking multiple sweeps, and want the sweep timing to work, put the turn continuous mode off. like pnax.continuous='off'

skrf.vi.vna.HP8510C.s11

HP8510C.**s11**

skrf.vi.vna.HP8510C.s12

HP8510C.**s12**

skrf.vi.vna.HP8510C.s21

HP8510C.**s21**

skrf.vi.vna.HP8510C.s22

HP8510C.**s22**

skrf.vi.vna.HP8510C.switch_terms

HP8510C.**switch_terms**

measures forward and reverse switch terms and returns them as a pair of one-port networks.

returns:

forward, reverse: a tuple of one ports holding forward and reverse switch terms.

see also: skrf.calibrationAlgorithms.unterminate_switch_terms

notes: thanks to dylan williams for making me aware of this, and providing the gpib commands in his statistical help

skrf.vi.vna.HP8510C.two_port

HP8510C.**two_port**

Initiates a sweep and returns a Network type represting the data.

if you are taking multiple sweeps, and want the sweep timing to work, put the turn continuous mode off. like pnax.continuous='off'

Methods

__init__

skrf.vi.vna.HP8510C.__init__

HP8510C.**__init__**(*address=16*, ***kwargs*)

skrf.vi.vna.HP8720

class skrf.vi.vna.**HP8720**(*address=16*, ***kwargs*)

Attributes

averaging	
continuous	
error	
frequency	
ifbw	
one_port	Initiates a sweep and returns a Network type represting the
s11	
s12	
s21	
s22	
switch_terms	measures forward and reverse switch terms and returns them as a
two_port	Initiates a sweep and returns a Network type represting the

skrf.vi.vna.HP8720.averaging

HP8720.**averaging**

skrf.vi.vna.HP8720.continuous

HP8720.**continuous**

skrf.vi.vna.HP8720.error

HP8720.**error**

skrf.vi.vna.HP8720.frequency

HP8720.**frequency**

skrf.vi.vna.HP8720.ifbw

HP8720.**ifbw**

skrf.vi.vna.HP8720.one_port

HP8720.**one_port**

Initiates a sweep and returns a Network type represting the data.

if you are taking multiple sweeps, and want the sweep timing to work, put the turn continuous mode off. like pnax.continuous='off'

skrf.vi.vna.HP8720.s11

HP8720.**s11**

skrf.vi.vna.HP8720.s12

HP8720.**s12**

skrf.vi.vna.HP8720.s21

HP8720.**s21**

skrf.vi.vna.HP8720.s22

HP8720.**s22**

skrf.vi.vna.HP8720.switch_terms

HP8720.**switch_terms**

measures forward and reverse switch terms and returns them as a pair of one-port networks.

returns:

forward, reverse: a tuple of one ports holding forward and reverse switch terms.

see also: skrf.calibrationAlgorithms.unterminate_switch_terms

notes: thanks to dylan williams for making me aware of this, and providing the gpib commands in his statistical help

skrf.vi.vna.HP8720.two_port

HP8720.**two_port**

Initiates a sweep and returns a Network type represting the data.

if you are taking multiple sweeps, and want the sweep timing to work, put the turn continuous mode off. like pnax.continuous='off'

Methods

`__init__`

skrf.vi.vna.HP8720.__init__

HP8720.**__init__**(*address=16*, ***kwargs*)

3.12.2 Spectrum Analyzers (skrf.vi.sa)

HP8500([address])	HP8500's series Spectrum Analyzers

skrf.vi.sa.HP8500

class skrf.vi.sa.**HP8500**(*address=18*, **args*, ***kwargs*)
HP8500's series Spectrum Analyzers

Examples

Get trace, and store in a Network object

```
>>> from skrf.vi.sa import HP
>>> my_sa = HP() # default address is 18
>>> trace = my_sa.get_ntwk()
```

Activate single sweep mode, get a trace, return to continuous sweep

```
>>> my_sa.single_sweep()
>>> my_sa.sweep()
>>> trace_a = my_sa.trace_a
>>> my_sa.cont_sweep()
```

Attributes

f_start	starting frequency
f_stop	stopping frequency
frequency	
get_ntwk	Get a trace and return the data in a Network format
trace_a	trace 'a'
trace_b	trace 'b'

skrf.vi.sa.HP8500.f_start

HP8500.**f_start**
starting frequency

skrf.vi.sa.HP8500.f_stop

HP8500.**f_stop**
stopping frequency

skrf.vi.sa.HP8500.frequency

HP8500.**frequency**

skrf.vi.sa.HP8500.get_ntwk

HP8500.**get_ntwk**

Get a trace and return the data in a `Network` format

This will save instrument stage to reg 1, activate single sweep mode, sweep, save data, then recal state from reg 1.

Returning the data in a the form of a `Network` allows all the plotting methods and IO functions of that class to be used. Not all the methods of Network make sense for this type of data (scalar), but we assume the user is knows this.

Parameters **trace** : ['a', 'b']

save trace 'a' or trace 'b'

goto_local : Boolean

Go to local mode after taking a sweep

***args,**kwargs :** :

passed to `__init__()`

skrf.vi.sa.HP8500.trace_a

HP8500.**trace_a**

trace 'a'

skrf.vi.sa.HP8500.trace_b

HP8500.**trace_b**

trace 'b'

Methods

`__init__`	Initializer
`cont_sweep`	Activate continuous sweep mode
`goto_local`	Switches from remote to local control
`recall_state`	Recall current state to a given register
`save_state`	Save current state to a given register
`single_sweep`	Activate single sweep mode
`sweep`	trigger a sweep, return when done

skrf.vi.sa.HP8500.__init__

HP8500.**__init__**(*address=18*, **args*, ***kwargs*)

Initializer

Parameters **address** : int

GPIB address

***args, **kwargs :** :

passed to `visa.GpibInstrument.__init__`

skrf.vi.sa.HP8500.cont_sweep

HP8500.**cont_sweep**()

Activate continuous sweep mode

skrf.vi.sa.HP8500.goto_local

HP8500.**goto_local**()

Switches from remote to local control

skrf.vi.sa.HP8500.recall_state

HP8500.**recall_state**(*reg_n*)

Recall current state to a given register

skrf.vi.sa.HP8500.save_state

HP8500.**save_state**(*reg_n=1*)

Save current state to a given register

skrf.vi.sa.HP8500.single_sweep

HP8500.**single_sweep**()

Activate single sweep mode

skrf.vi.sa.HP8500.sweep

HP8500.**sweep**()

trigger a sweep, return when done

3.12.3 Stages (`skrf.vi.stages`)

`ESP300`([address, current_axis, ...])	Newport Universal Motion Controller/Driver Model ESP300

skrf.vi.stages.ESP300

class skrf.vi.stages.**ESP300**(*address=1*, *current_axis=1*, *always_wait_for_stop=True*, *delay=0*, ***kwargs*)

Newport Universal Motion Controller/Driver Model ESP300 all axis control commands are sent to the number axis given by the local variable self.current_axis. An example usage

```
from skrf.vi.stages import ESP300
esp = ESP300()
esp.current_axis = 1
esp.position = 10
print esp.position
```

Attributes

UNIT_DICT	
acceleration	
current_axis	current axis used in all subsequent commands
deceleration	
error_message	
home	
motor_on	
position	
position_relative	
units	
velocity	the velocity of current axis

skrf.vi.stages.ESP300.UNIT_DICT

ESP300.**UNIT_DICT = {'micro inches': 6, 'inches': 4, 'micrometer': 3, 'milliradian': 10, 'millimeter': 2, 'milli inches': 5, 'enc**

skrf.vi.stages.ESP300.acceleration

ESP300.**acceleration**

skrf.vi.stages.ESP300.current_axis

ESP300.**current_axis**
current axis used in all subsequent commands

skrf.vi.stages.ESP300.deceleration

ESP300.**deceleration**

skrf.vi.stages.ESP300.error_message

ESP300.**error_message**

skrf.vi.stages.ESP300.home

ESP300.**home**

skrf.vi.stages.ESP300.motor_on

ESP300.**motor_on**

skrf.vi.stages.ESP300.position

ESP300.**position**

skrf.vi.stages.ESP300.position_relative

ESP300.**position_relative**

skrf.vi.stages.ESP300.units

ESP300.**units**

skrf.vi.stages.ESP300.velocity

ESP300.**velocity**
the velocity of current axis

Methods

__init__	Initializer
send_stop	
wait_for_stop	

skrf.vi.stages.ESP300.__init__

ESP300.**__init__**(*address=1*, *current_axis=1*, *always_wait_for_stop=True*, *delay=0*, ***kwargs*)
Initializer

Parameters **address** : int

Gpib address

current_axis : int

number of current axis

always_wait_for_stop : Boolean

wait for stage to stop before returning control to calling program

****kwargs :** :

passed to GpibInstrument initializer

skrf.vi.stages.ESP300.send_stop

ESP300.**send_stop**()

skrf.vi.stages.ESP300.wait_for_stop

ESP300.**wait_for_stop**()

3.13 Indices and tables

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

S

skrf.calibration, 165
skrf.calibration.calibration, 165
skrf.calibration.calibrationAlgorithms, 172
skrf.calibration.calibrationFunctions, 176
skrf.constants, 155
skrf.frequency, 53
skrf.io, 158
skrf.io.general, 158
skrf.io.touchstone, 162
skrf.mathFunctions, 144
skrf.media, 184
skrf.network, 57
skrf.networkSet, 132
skrf.plotting, 138
skrf.tlineFunctions, 147
skrf.util, 156
skrf.vi, 254
skrf.vi.sa, 262
skrf.vi.stages, 264
skrf.vi.vna, 254

INDEX

Symbols

__init__() (skrf.calibration.calibration.Calibration method), 168, 180
__init__() (skrf.frequency.Frequency method), 56
__init__() (skrf.io.touchstone.Touchstone method), 163
__init__() (skrf.media.cpw.CPW method), 229
__init__() (skrf.media.distributedCircuit.DistributedCircuit method), 201
__init__() (skrf.media.freespace.Freespace method), 243
__init__() (skrf.media.media.Media method), 187
__init__() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 215
__init__() (skrf.network.Network method), 72
__init__() (skrf.networkSet.NetworkSet method), 134
__init__() (skrf.vi.sa.HP8500 method), 263
__init__() (skrf.vi.stages.ESP300 method), 266
__init__() (skrf.vi.vna.HP8510C method), 260
__init__() (skrf.vi.vna.HP8720 method), 262
__init__() (skrf.vi.vna.PNAX method), 255
__init__() (skrf.vi.vna.ZVA40 method), 257

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), 176
acceleration (skrf.vi.stages.ESP300 attribute), 265
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
add_trace() (skrf.vi.vna.ZVA40 method), 257
alpha_conductor (skrf.media.cpw.CPW attribute), 227
apply_cal() (skrf.calibration.calibration.Calibration method), 168, 180
apply_cal_to_all_in_dir() (skrf.calibration.calibration.Calibration method), 169, 180
average() (in module skrf.network), 131
averaging (skrf.vi.vna.HP8510C attribute), 258
averaging (skrf.vi.vna.HP8720 attribute), 260

B

biased_error() (skrf.calibration.calibration.Calibration method), 169, 181

C

caled_ntwk_sets (skrf.calibration.calibration.Calibration attribute), 166, 178
caled_ntwks (skrf.calibration.calibration.Calibration attribute), 166, 178
Calibration (class in skrf.calibration.calibration), 165, 177
calibration_algorithm_dict (skrf.calibration.calibration.Calibration attribute), 166, 178
capacitor() (skrf.media.cpw.CPW method), 230
capacitor() (skrf.media.distributedCircuit.DistributedCircuit method), 202
capacitor() (skrf.media.freespace.Freespace method), 244
capacitor() (skrf.media.media.Media method), 188
capacitor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 216
cartesian_product_calibration_set() (in module skrf.calibration.calibrationFunctions), 176
cascade() (in module skrf.network), 119
center (skrf.frequency.Frequency attribute), 54
characteristic_impedance (skrf.media.cpw.CPW attribute), 228
characteristic_impedance (skrf.media.distributedCircuit.DistributedCircuit attribute), 199
characteristic_impedance (skrf.media.freespace.Freespace attribute),

241
characteristic_impedance (skrf.media.media.Media attribute), 185
characteristic_impedance (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 213
coefs (skrf.calibration.calibration.Calibration attribute), 166, 178
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
cont_sweep() (skrf.vi.sa.HP8500 method), 264
continuous (skrf.vi.vna.HP8510C attribute), 258
continuous (skrf.vi.vna.HP8720 attribute), 260
continuous (skrf.vi.vna.PNAX attribute), 255
continuous (skrf.vi.vna.ZVA40 attribute), 256
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), 227
current_axis (skrf.vi.stages.ESP300 attribute), 265

D

db_2_np() (in module skrf.mathFunctions), 146
de_embed() (in module skrf.network), 119
deceleration (skrf.vi.stages.ESP300 attribute), 265
degree_2_radian() (in module skrf.mathFunctions), 145
delay_load() (skrf.media.cpw.CPW method), 230
delay_load() (skrf.media.distributedCircuit.DistributedCircuit method), 202
delay_load() (skrf.media.freespace.Freespace method), 244
delay_load() (skrf.media.media.Media method), 188
delay_load() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 217
delay_open() (skrf.media.cpw.CPW method), 231
delay_open() (skrf.media.distributedCircuit.DistributedCircuit method), 203
delay_open() (skrf.media.freespace.Freespace method), 245
delay_open() (skrf.media.media.Media method), 189
delay_open() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 217
delay_short() (skrf.media.cpw.CPW method), 232
delay_short() (skrf.media.distributedCircuit.DistributedCircuit method), 203
delay_short() (skrf.media.freespace.Freespace method), 245
delay_short() (skrf.media.media.Media method), 189
delay_short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 218
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), 198

E

eight_term_2_one_port_coefs() (in module skrf.calibration.calibrationAlgorithms), 176
electrical_length() (skrf.media.cpw.CPW method), 232
electrical_length() (skrf.media.distributedCircuit.DistributedCircuit method), 204
electrical_length() (skrf.media.freespace.Freespace method), 246
electrical_length() (skrf.media.media.Media method), 190
electrical_length() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 218
electrical_length_2_distance() (in module skrf.tlineFunctions), 151
element_wise_method() (skrf.networkSet.NetworkSet method), 135
ep (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 213
ep_re (skrf.media.cpw.CPW attribute), 228
error (skrf.vi.vna.HP8510C attribute), 258
error (skrf.vi.vna.HP8720 attribute), 260
error (skrf.vi.vna.ZVA40 attribute), 256
error_message (skrf.vi.stages.ESP300 attribute), 265
error_ntwk (skrf.calibration.calibration.Calibration attribute), 166, 178
ESP300 (class in skrf.vi.stages), 264

F

f (skrf.frequency.Frequency attribute), 54
f (skrf.network.Network attribute), 62
f_scaled (skrf.frequency.Frequency attribute), 54
f_start (skrf.vi.sa.HP8500 attribute), 262
f_stop (skrf.vi.sa.HP8500 attribute), 262
fdata (skrf.vi.vna.ZVA40 attribute), 256
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), 240
Frequency (class in skrf.frequency), 53
frequency (skrf.network.Network attribute), 62

frequency (skrf.vi.sa.HP8500 attribute), 263
frequency (skrf.vi.vna.HP8510C attribute), 258
frequency (skrf.vi.vna.HP8720 attribute), 260
frequency (skrf.vi.vna.PNAX attribute), 255
frequency (skrf.vi.vna.ZVA40 attribute), 256
from_csv() (skrf.media.cpw.CPW class method), 232
from_csv() (skrf.media.distributedCircuit.DistributedCircuit class method), 204
from_csv() (skrf.media.freespace.Freespace class method), 246
from_csv() (skrf.media.media.Media class method), 190
from_csv() (skrf.media.rectangularWaveguide.RectangularWaveguide class method), 219
from_f() (skrf.frequency.Frequency class method), 57
from_Media() (skrf.media.distributedCircuit.DistributedCircuit class method), 204
from_Media() (skrf.media.freespace.Freespace class method), 246
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), 169, 181

G

gamma() (skrf.media.cpw.CPW method), 233
gamma() (skrf.media.distributedCircuit.DistributedCircuit method), 204
gamma() (skrf.media.freespace.Freespace method), 246
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_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_ntwk (skrf.vi.sa.HP8500 attribute), 263
get_sparameter_arrays() (skrf.io.touchstone.Touchstone method), 163
get_sparameter_data() (skrf.io.touchstone.Touchstone method), 164
get_sparameter_names() (skrf.io.touchstone.Touchstone method), 164
goto_local() (skrf.vi.sa.HP8500 method), 264
guess_length_of_delay_short() (skrf.media.cpw.CPW method), 233
guess_length_of_delay_short() (skrf.media.distributedCircuit.DistributedCircuit method), 205
guess_length_of_delay_short() (skrf.media.freespace.Freespace method), 247
guess_length_of_delay_short() (skrf.media.media.Media method), 190
guess_length_of_delay_short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 219

H

hfss_touchstone_2_gamma_z0() (in module skrf.io.touchstone), 164
hfss_touchstone_2_media() (in module skrf.io.touchstone), 165
home (skrf.vi.stages.ESP300 attribute), 265
HP8500 (class in skrf.vi.sa), 262
HP8510C (class in skrf.vi.vna), 258
HP8720 (class in skrf.vi.vna), 260

I

ifbw (skrf.vi.vna.HP8720 attribute), 260
impedance_mismatch() (skrf.media.cpw.CPW method), 233
impedance_mismatch() (skrf.media.distributedCircuit.DistributedCircuit method), 205
impedance_mismatch() (skrf.media.freespace.Freespace method), 247
impedance_mismatch() (skrf.media.media.Media method), 190
impedance_mismatch() (skrf.media.rectangularWaveguide.RectangularWav method), 219
inductor() (skrf.media.cpw.CPW method), 233
inductor() (skrf.media.distributedCircuit.DistributedCircuit method), 205
inductor() (skrf.media.freespace.Freespace method), 247
inductor() (skrf.media.media.Media method), 191
inductor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 220
initiate() (skrf.vi.vna.ZVA40 method), 257
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), 213
k1 (skrf.media.cpw.CPW attribute), 228
K_ratio (skrf.media.cpw.CPW attribute), 227
kc (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 213
kx (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 213
ky (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 214
kz() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 220

L

labelXAxis() (skrf.frequency.Frequency method), 57
legend_off() (in module skrf.plotting), 143
line() (skrf.media.cpw.CPW method), 234
line() (skrf.media.distributedCircuit.DistributedCircuit method), 206
line() (skrf.media.freespace.Freespace method), 248
line() (skrf.media.media.Media method), 191
line() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 220
load() (skrf.media.cpw.CPW method), 234
load() (skrf.media.distributedCircuit.DistributedCircuit method), 206
load() (skrf.media.freespace.Freespace method), 248
load() (skrf.media.media.Media method), 192
load() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 221
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), 235
match() (skrf.media.distributedCircuit.DistributedCircuit method), 207
match() (skrf.media.freespace.Freespace method), 249
match() (skrf.media.media.Media method), 192
match() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 221
mean_residuals() (skrf.calibration.calibration.Calibration method), 169, 181
mean_s_db (skrf.networkSet.NetworkSet attribute), 133
Media (class in skrf.media.media), 185
motor_on (skrf.vi.stages.ESP300 attribute), 266
mu (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 214
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
network (skrf.vi.vna.PNAX attribute), 255
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), 167, 178
nports (skrf.network.Network attribute), 63
nstandards (skrf.calibration.calibration.Calibration attribute), 167, 178
nudge() (skrf.network.Network method), 77, 132
null() (in module skrf.mathFunctions), 147
number_of_ports (skrf.network.Network attribute), 63

O

one_port (skrf.vi.vna.HP8510C attribute), 259
one_port (skrf.vi.vna.HP8720 attribute), 261
one_port (skrf.vi.vna.ZVA40 attribute), 256
one_port() (in module skrf.calibration.calibrationAlgorithms), 172
one_port_nls() (in module skrf.calibration.calibrationAlgorithms), 173
open() (skrf.media.cpw.CPW method), 235
open() (skrf.media.distributedCircuit.DistributedCircuit method), 207
open() (skrf.media.freespace.Freespace method), 249
open() (skrf.media.media.Media method), 193
open() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 222
output_from_cal (skrf.calibration.calibration.Calibration attribute), 167, 178

P

parameterized_self_calibration() (in module skrf.calibration.calibrationAlgorithms), 174
parameterized_self_calibration_nls() (in module skrf.calibration.calibrationAlgorithms), 175
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), 169, 181
plot_complex_polar() (in module skrf.plotting), 142
plot_complex_rectangular() (in module skrf.plotting), 141
plot_errors() (skrf.calibration.calibration.Calibration method), 169, 181
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), 170, 182
plot_residuals_db() (skrf.calibration.calibration.Calibration method), 170, 182
plot_residuals_mag() (skrf.calibration.calibration.Calibration method), 170, 182
plot_residuals_smith() (skrf.calibration.calibration.Calibration method), 170, 182
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), 170, 182
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
PNAX (class in skrf.vi.vna), 254
position (skrf.vi.stages.ESP300 attribute), 266
position_relative (skrf.vi.stages.ESP300 attribute), 266
propagation_constant (skrf.media.cpw.CPW attribute), 228
propagation_constant (skrf.media.distributedCircuit.DistributedCircuit attribute), 199
propagation_constant (skrf.media.freespace.Freespace attribute), 241
propagation_constant (skrf.media.media.Media attribute), 185
propagation_constant (skrf.media.rectangularWaveguide.RectangularWaveg attribute), 214
propagation_impedance_2_distributed_circuit() (in module 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_touchstone() (skrf.network.Network method), 115
recall_state() (skrf.vi.sa.HP8500 method), 264
RectangularWaveguide (class in skrf.media.rectangularWaveguide), 212
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), 167, 179
residuals (skrf.calibration.calibration.Calibration attribute), 167, 179
resistor() (skrf.media.cpw.CPW method), 235
resistor() (skrf.media.distributedCircuit.DistributedCircuit method), 207
resistor() (skrf.media.freespace.Freespace method), 249
resistor() (skrf.media.media.Media method), 193
resistor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 222
run() (skrf.calibration.calibration.Calibration method), 170, 182

S

s (skrf.network.Network attribute), 64
s11 (skrf.network.Network attribute), 64
s11 (skrf.vi.vna.HP8510C attribute), 259
s11 (skrf.vi.vna.HP8720 attribute), 261
s11 (skrf.vi.vna.ZVA40 attribute), 256
s12 (skrf.network.Network attribute), 64
s12 (skrf.vi.vna.HP8510C attribute), 259
s12 (skrf.vi.vna.HP8720 attribute), 261
s21 (skrf.network.Network attribute), 64
s21 (skrf.vi.vna.HP8510C attribute), 259
s21 (skrf.vi.vna.HP8720 attribute), 261
s22 (skrf.network.Network attribute), 64
s22 (skrf.vi.vna.HP8510C attribute), 259
s22 (skrf.vi.vna.HP8720 attribute), 261
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
save_state() (skrf.vi.sa.HP8500 method), 264
scalar2Complex() (in module skrf.mathFunctions), 146
sdata (skrf.vi.vna.ZVA40 attribute), 257
send_stop() (skrf.vi.stages.ESP300 method), 266
set_active_trace() (skrf.vi.vna.ZVA40 method), 257
set_wise_function() (skrf.networkSet.NetworkSet method), 137
short() (skrf.media.cpw.CPW method), 236
short() (skrf.media.distributedCircuit.DistributedCircuit method), 208
short() (skrf.media.freespace.Freespace method), 250
short() (skrf.media.media.Media method), 193
short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 222
shunt() (skrf.media.cpw.CPW method), 236
shunt() (skrf.media.distributedCircuit.DistributedCircuit method), 208
shunt() (skrf.media.freespace.Freespace method), 250
shunt() (skrf.media.media.Media method), 194
shunt() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 223
shunt_capacitor() (skrf.media.cpw.CPW method), 236
shunt_capacitor() (skrf.media.distributedCircuit.DistributedCircuit method), 208
shunt_capacitor() (skrf.media.freespace.Freespace method), 250
shunt_capacitor() (skrf.media.media.Media method), 194
shunt_capacitor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 223
shunt_delay_load() (skrf.media.cpw.CPW method), 237
shunt_delay_load() (skrf.media.distributedCircuit.DistributedCircuit method), 209
shunt_delay_load() (skrf.media.freespace.Freespace method), 251
shunt_delay_load() (skrf.media.media.Media method), 194
shunt_delay_load() (skrf.media.rectangularWaveguide.RectangularWavegui method), 223
shunt_delay_open() (skrf.media.cpw.CPW method), 237
shunt_delay_open() (skrf.media.distributedCircuit.DistributedCircuit method), 209
shunt_delay_open() (skrf.media.freespace.Freespace method), 251
shunt_delay_open() (skrf.media.media.Media method), 195
shunt_delay_open() (skrf.media.rectangularWaveguide.RectangularWavegu method), 224
shunt_delay_short() (skrf.media.cpw.CPW method), 237
shunt_delay_short() (skrf.media.distributedCircuit.DistributedCircuit method), 209
shunt_delay_short() (skrf.media.freespace.Freespace method), 251
shunt_delay_short() (skrf.media.media.Media method), 195
shunt_delay_short() (skrf.media.rectangularWaveguide.RectangularWavegu method), 224
shunt_inductor() (skrf.media.cpw.CPW method), 238
shunt_inductor() (skrf.media.distributedCircuit.DistributedCircuit method), 210

shunt_inductor() (skrf.media.freespace.Freespace method), 252
shunt_inductor() (skrf.media.media.Media method), 195
shunt_inductor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 224
signature() (skrf.networkSet.NetworkSet method), 137
single_sweep() (skrf.vi.sa.HP8500 method), 264
skin_depth() (in module skrf.tlineFunctions), 155
skrf.calibration (module), 165
skrf.calibration.calibration (module), 165
skrf.calibration.calibrationAlgorithms (module), 172
skrf.calibration.calibrationFunctions (module), 176
skrf.constants (module), 155
skrf.frequency (module), 53
skrf.io (module), 158
skrf.io.general (module), 158
skrf.io.touchstone (module), 162
skrf.mathFunctions (module), 144
skrf.media (module), 184
skrf.network (module), 57
skrf.networkSet (module), 132
skrf.plotting (module), 138
skrf.tlineFunctions (module), 147
skrf.util (module), 156
skrf.vi (module), 254
skrf.vi.sa (module), 262
skrf.vi.stages (module), 264
skrf.vi.vna (module), 254
smith() (in module skrf.plotting), 138
span (skrf.frequency.Frequency attribute), 55
splitter() (skrf.media.cpw.CPW method), 238
splitter() (skrf.media.distributedCircuit.DistributedCircuit method), 210
splitter() (skrf.media.freespace.Freespace method), 252
splitter() (skrf.media.media.Media method), 196
splitter() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 225
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
surface_resistivity() (in module skrf.tlineFunctions), 155
sweep() (skrf.vi.sa.HP8500 method), 264
sweep() (skrf.vi.vna.ZVA40 method), 257
switch_terms (skrf.vi.vna.HP8510C attribute), 259
switch_terms (skrf.vi.vna.HP8720 attribute), 261

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), 239
tee() (skrf.media.distributedCircuit.DistributedCircuit method), 210
tee() (skrf.media.freespace.Freespace method), 252
tee() (skrf.media.media.Media method), 196
tee() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 225
theta() (in module skrf.tlineFunctions), 148
theta_2_d() (skrf.media.cpw.CPW method), 239
theta_2_d() (skrf.media.distributedCircuit.DistributedCircuit method), 211
theta_2_d() (skrf.media.freespace.Freespace method), 253
theta_2_d() (skrf.media.media.Media method), 196
theta_2_d() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 225
thru() (skrf.media.cpw.CPW method), 239
thru() (skrf.media.distributedCircuit.DistributedCircuit method), 211
thru() (skrf.media.freespace.Freespace method), 253
thru() (skrf.media.media.Media method), 197
thru() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 226
total_error() (skrf.calibration.calibration.Calibration method), 171, 183
Touchstone (class in skrf.io.touchstone), 162
trace_a (skrf.vi.sa.HP8500 attribute), 263
trace_b (skrf.vi.sa.HP8500 attribute), 263
Ts (skrf.calibration.calibration.Calibration attribute), 166, 178
two_port (skrf.vi.vna.HP8510C attribute), 259
two_port (skrf.vi.vna.HP8720 attribute), 261
two_port() (in module skrf.calibration.calibrationAlgorithms), 173
type (skrf.calibration.calibration.Calibration attribute), 167, 179

U

unbiased_error() (skrf.calibration.calibration.Calibration method), 171, 183
uncertainty_ntwk_triplet() (skrf.networkSet.NetworkSet method), 137
uncertainty_per_standard() (skrf.calibration.calibration.Calibration method), 171, 183
unit (skrf.frequency.Frequency attribute), 55
unit_dict (skrf.frequency.Frequency attribute), 55
UNIT_DICT (skrf.vi.stages.ESP300 attribute), 265
units (skrf.vi.stages.ESP300 attribute), 266
unterminate_switch_terms() (in module skrf.calibration.calibrationAlgorithms), 176
unwrap_rad() (in module skrf.mathFunctions), 145
update_trace_list() (skrf.vi.vna.ZVA40 method), 257

upload_cal_data() (skrf.vi.vna.ZVA40 method), 258

V

velocity (skrf.vi.stages.ESP300 attribute), 266

W

w (skrf.frequency.Frequency attribute), 56
wait() (skrf.vi.vna.ZVA40 method), 258
wait_for_stop() (skrf.vi.stages.ESP300 method), 267
white_gaussian_polar() (skrf.media.cpw.CPW method), 240
white_gaussian_polar() (skrf.media.distributedCircuit.DistributedCircuit method), 211
white_gaussian_polar() (skrf.media.freespace.Freespace method), 253
white_gaussian_polar() (skrf.media.media.Media method), 197
white_gaussian_polar() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 226
write() (in module skrf.io.general), 160
write() (skrf.calibration.calibration.Calibration method), 172, 184
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), 240
write_csv() (skrf.media.distributedCircuit.DistributedCircuit method), 212
write_csv() (skrf.media.freespace.Freespace method), 254
write_csv() (skrf.media.media.Media method), 198
write_csv() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 227
write_touchstone() (skrf.network.Network method), 116

Y

Y (skrf.media.distributedCircuit.DistributedCircuit attribute), 199
Y (skrf.media.freespace.Freespace attribute), 241
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), 199
Z (skrf.media.freespace.Freespace attribute), 241
z (skrf.network.Network attribute), 68
z0 (skrf.media.cpw.CPW attribute), 228
z0 (skrf.media.distributedCircuit.DistributedCircuit attribute), 200
z0 (skrf.media.freespace.Freespace attribute), 242
z0 (skrf.media.media.Media attribute), 186
z0 (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 214
z0 (skrf.network.Network attribute), 69
Z0() (skrf.media.cpw.CPW method), 229
Z0() (skrf.media.distributedCircuit.DistributedCircuit method), 201
Z0() (skrf.media.freespace.Freespace method), 243
Z0() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 215
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
ZVA40 (class in skrf.vi.vna), 255