



scikit-rf

Object Oriented RF Engineering

scikit-rf Documentation

Release 0.1

alex arsenovic

September 26, 2012

CONTENTS

1	Tutorials	3
1.1	Installation	3
1.2	Introduction	4
1.3	Plotting	8
1.4	Calibration	13
1.5	Circuit Design	17
1.6	Uncertainty Estimation	20
1.7	Developing skrf	22
2	Examples	25
2.1	One-Port Calibration	25
2.2	Two-Port Calibration	26
2.3	VNA Noise Analysis	27
2.4	Circuit Design: Single Stub Matching Network	28
3	Reference	35
3.1	Major Classes	35
3.2	Modules	35
3.3	Packages	90
	Bibliography	175
	Python Module Index	177
	Python Module Index	179
	Index	181

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

TUTORIALS

1.1 Installation

Contents

- Installation
 - Introduction
 - skrf 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 [Python](#) development, you may want to install a pre-built scientific python IDE like [pythonxy](#). This will install all requirements, as well as provide a nice environment to get started in. If you dont want use [pythonxy](#), you see [Requirements](#).

Note: If you want to use skrf for instrument control you will need to install pyvisa. You may also be interested in Pythics, which provides a simple way to build 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

They can all be found at <http://scikit-rf.org/download/>

If you dont know how to install a python module and dont care to learn how, you want the windows installer. Otherwise, I recommend the python source package because examples, documentation, and installation instructions are provided with the the python package.

The current version can be accessed through [github](#). This is mainly of interest for developers, and is not stable most of the time.

1.1.3 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
```

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
 - Creating Networks
 - Basic Network Properties
 - Network Operators
 - * Element-wise Operations
 - * Cascading and Embedding Operations
 - Connecting Multi-ports
 - Sub-Networks
 - Convenience Functions
 - References

This is a brief introduction to skrf, aimed at those who are familiar with python. If you are unfamiliar with python, please see scipy's [Getting Started](#) . This tutorial is most easily followed by using the `ipython` shell with the `-pylab` flag.

Note: All of the scripts and touchstone files used in these tutorials are provided along with this source code in the directory `doc/source/pyplots/` (relative to the scikit-rf root).

1.2.1 Creating Networks

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:


```
>>> import skrf as rf
```

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

skrf provides an object for a n-port microwave `Network`. Most commonly, a `Network` is constructed from data stored in a touchstone files, like so

```
>>> short = rf.Network('short.slp')
>>> delay_short = rf.Network('delay_short.slp')
```

Once created, the `Network` object will print out a short description if entered onto the command line:

```
>>> short
1-Port Network. 75-110 GHz. 201 points. z0=[ 50.]
```

1.2.2 Basic Network Properties

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

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

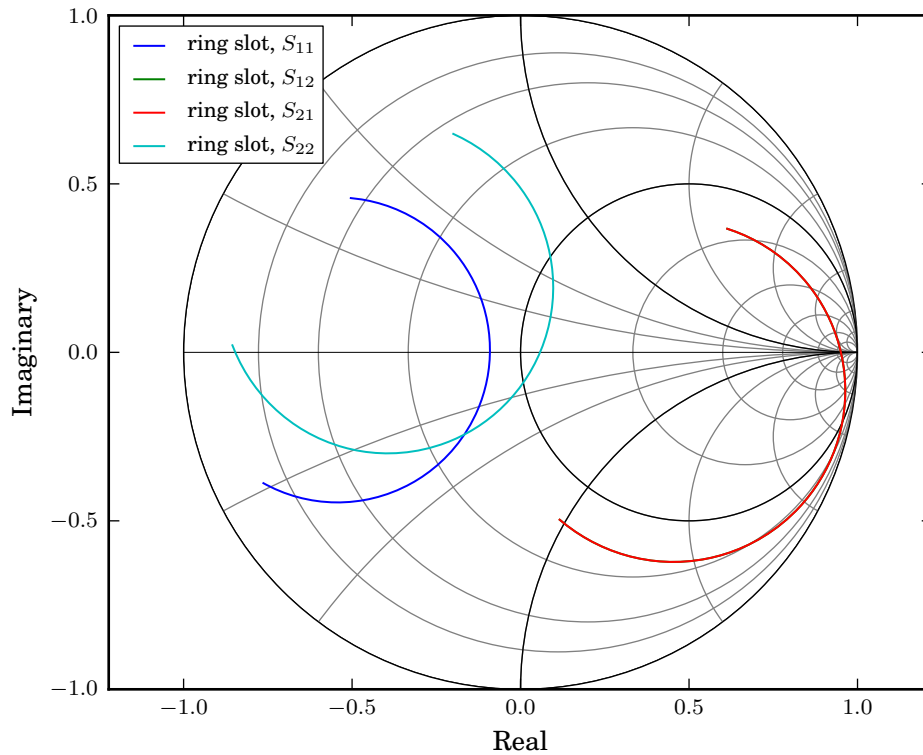
These properties are represented internally as complex `numpy.ndarray`'s, (and the indexing starts at 0!). The `Network` class has numerous other properties and methods which can be found in the `Network` docs. If you are using IPython, then these properties and methods can be 'tabbed' out on the command line. Amongst other things, the methods of the `Network` class provide convenient ways to plot components of the s-parameters, below is a short list of common plotting commands,

- `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

For example, to create a 2-port `Network` from a touchstone file, and then plot all s-parameters on the Smith Chart.

```
import pylab
import skrf as rf

# create a Network type from a touchstone file
ring_slot = rf.Network('ring_slot.s2p')
ring_slot.plot_s_smith()
pylab.show()
```



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

1.2.3 Network Operators

Element-wise Operations

Element-wise mathematical operations on the scattering parameter matrices are accessible through overloaded operators:

```
>>> short + delay_short
>>> short - delay_short
>>> short / delay_short
>>> short * delay_short
```

All of these operations return `Network` types, so the methods and properties of a `Network` are available on the result. For example, the difference operation ('-') can be used to calculate the complex distance between two networks

```
>>> difference = (short - delay_short)
```

The result is stored in the returned `Network` object's `Network.s` parameter matrix. So, to plot the magnitude of the complex difference between the networks `short` and `delay_short`:

```
>>> (short - delay_short).plot_s_mag()
```

Another common application is calculating the phase difference using the division operator. This can be done

```
>>> (delay_short/short).plot_s_deg()
```

Cascading and Embedding Operations

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

```
>>> line = rf.Network('line.s2p')
>>> short = rf.Network('short.s1p')
>>> delay_short = line ** short
```

De-embedding can be accomplished either by using the `de_embed()` function, or by cascading the *inverse* of a network. The inverse of a network is accessible through the property `Network.inv`. So, to de-embed the *short* from *delay_short*:

```
>>> short = line.inv ** delay_short
```

1.2.4 Connecting Multi-ports

skrf supports the connection of arbitrary ports of N-port networks. It accomplishes this using an algorithm called sub-network growth¹. This algorithm, which is available through the function `connect()`, takes into account port impedances. Terminating one port of a ideal 3-way splitter can be done like so:

```
>>> tee = rf.Network('tee.s3p')
>>> delay_short = rf.Network('delay_short.s1p')
```

to connect port '1' of the tee, to port 0 of the delay short:

```
>>> terminated_tee = rf.connect(tee,1,delay_short,0)
```

1.2.5 Sub-Networks

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

```
>>> port1_return = line.s11
>>> port1_insertion = line.s21
```

1.2.6 Convenience Functions

Frequently there is an entire directory of touchstone files that need to be analyzed. The function `load_all_touchstones()` is meant deal with this scenario. It takes a string representing the directory, and returns a dictionary type with keys equal to the touchstone filenames, and values equal to `Network` types:

```
>>> ntwk_dict = rf.load_all_touchstones('.')
{'delay_short': 1-Port Network. 75-110 GHz. 201 points. z0=[ 50.],
'line': 2-Port Network. 75-110 GHz. 201 points. z0=[ 50. 50.],
'ring slot': 2-Port Network. 75-110 GHz. 201 points. z0=[ 50. 50.],
'short': 1-Port Network. 75-110 GHz. 201 points. z0=[ 50.]}
```

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>

Other convenient functions, and pre-initialized objects are located in the `convenience` module

1.2.7 References

1.3 Plotting

Contents

- Plotting
 - Magnitude
 - Phase
 - Smith Chart
 - Multiple S-parameters
 - Comparing with Simulation
 - Saving Plots

This tutorial illustrates how to create common plots associated with microwave networks. The plotting functions are implemented as methods of the `Network` class, which is provided by the `skrf.network` module. Below is a list of the some of the plotting functions of network s-parameters,

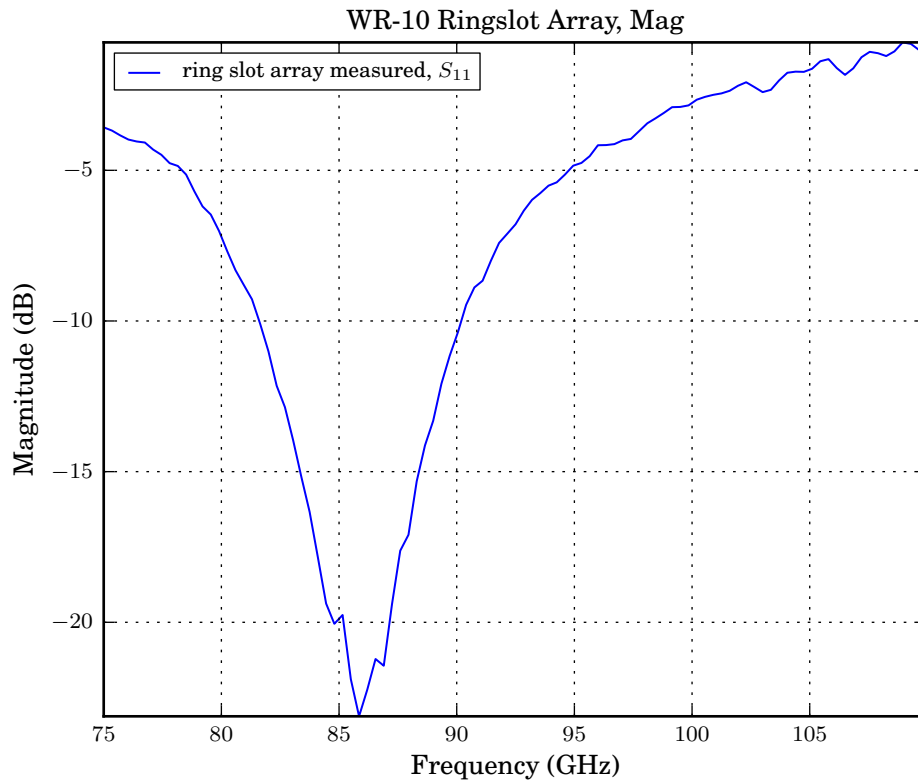
- `Network.plot_s_re()`
- `Network.plot_s_im()`
- `Network.plot_s_mag()`
- `Network.plot_s_db()`
- `Network.plot_s_deg()`
- `Network.plot_s_deg_unwrapped()`
- `Network.plot_s_rad()`
- `Network.plot_s_rad_unwrapped()`
- `Network.plot_s_smith()`
- `Network.plot_s_complex()`

1.3.1 Magnitude

```
import pylab
import skrf as rf

# create a Network type from a touchstone file of a horn antenna
ring_slot= rf.Network('ring slot array measured.slp')

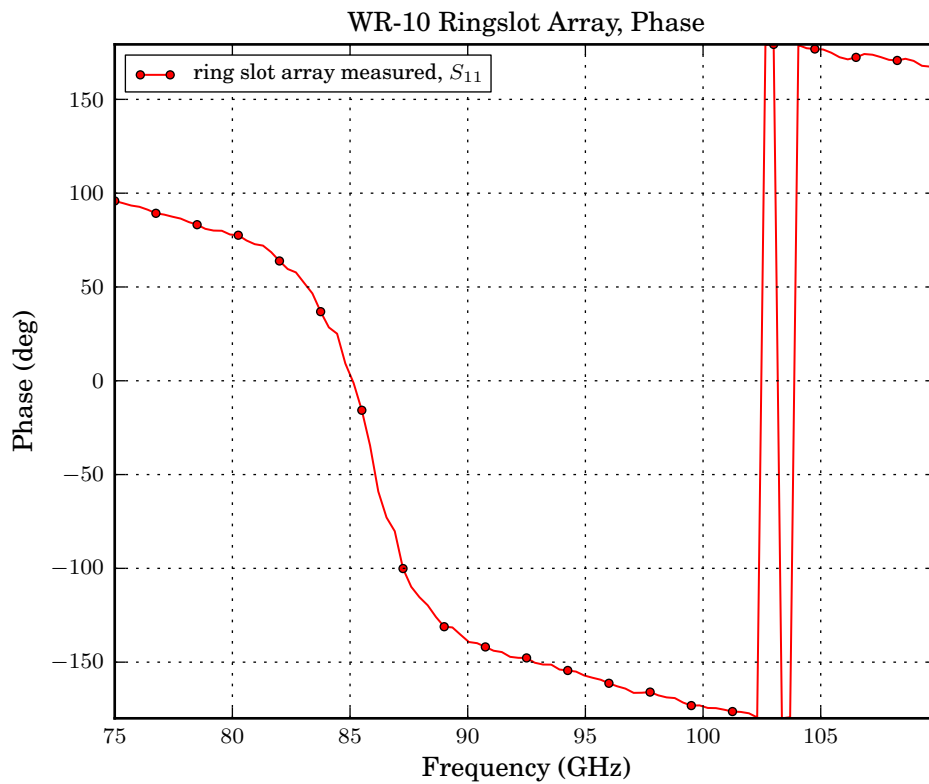
# plot magnitude (in db) of S11
pylab.figure(1)
pylab.title('WR-10 Ringslot Array, Mag')
ring_slot.plot_s_db(m=0,n=0) # m,n are S-Matrix indecies
# show the plots
pylab.show()
```



1.3.2 Phase

```
import pylab
import skrf as rf

ring_slot= rf.Network('ring slot array measured.slp')
pylab.figure(1)
pylab.title('WR-10 Ringslot Array, Phase')
# kwargs given to plot commands are passed through to the pylab.plot
# command
ring_slot.plot_s_deg(m=0,n=0, color='r', markevery=5, marker='o')
pylab.show()
```

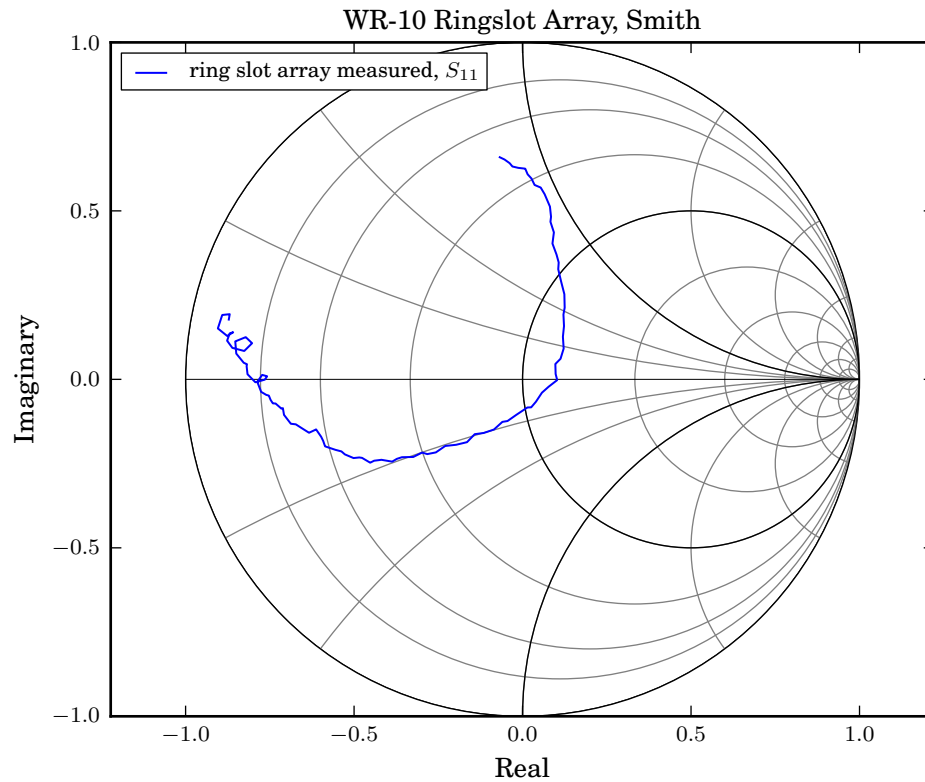


1.3.3 Smith Chart

```
import pylab
import skrf as rf

ring_slot= rf.Network('ring slot array measured.slp')

pylab.figure(1)
pylab.title('WR-10 Ringslot Array, Smith')
ring_slot.plot_s_smith(m=0,n=0) # m,n are S-Matrix indecies
pylab.show()
```



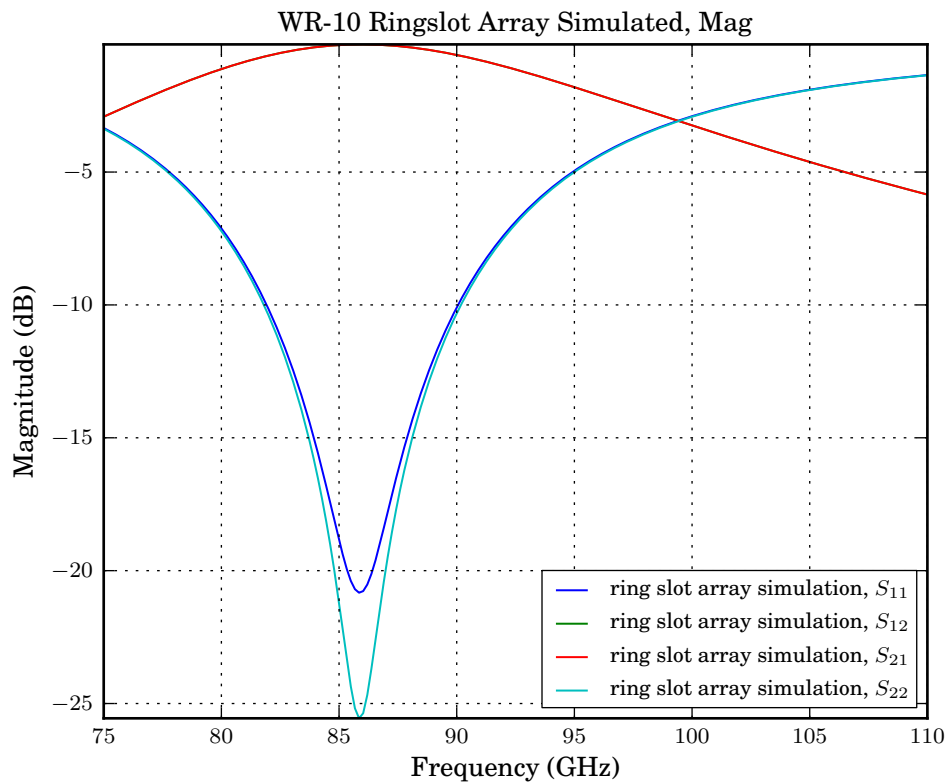
The Smith Chart can be also drawn independently of any `Network` object through the `smith()` function. This function also allows admittance contours can also be drawn.

1.3.4 Multiple S-parameters

```
import pylab
import skrf as rf

# from the extension you know this is a 2-port network
ring_slot= rf.Network('ring slot array simulation.s2p')

pylab.figure(1)
pylab.title('WR-10 Ringslot Array Simulated, Mag')
# if no indecies are passed to the plot command it will plot all
# available s-parameters
ring_slot.plot_s_db()
pylab.show()
```

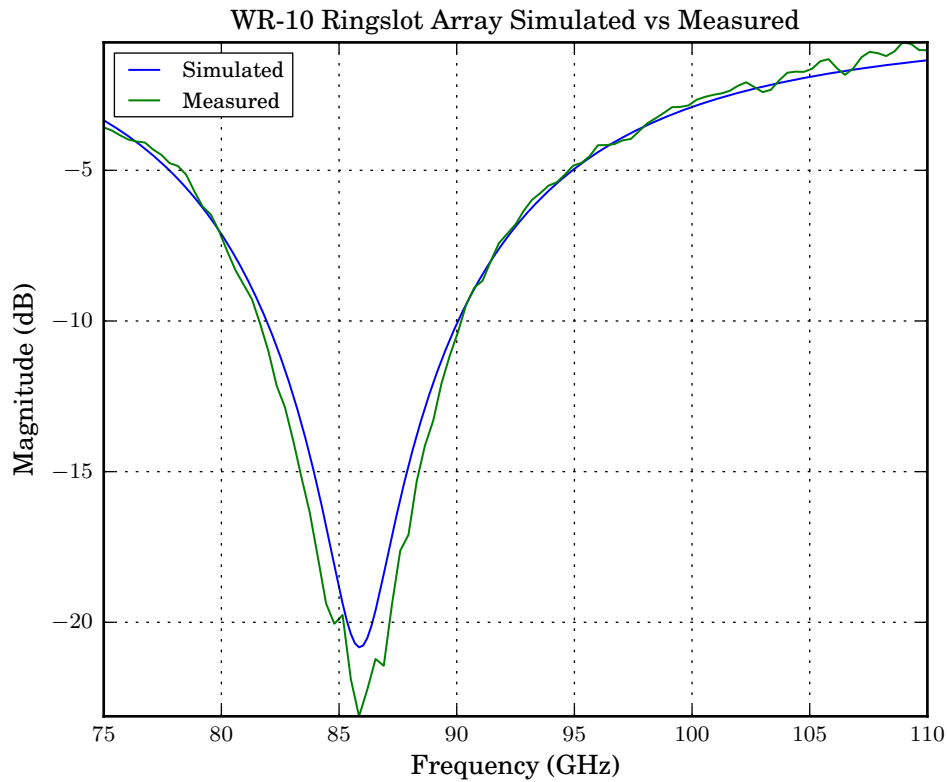


1.3.5 Comparing with Simulation

```
import pylab
import skrf as rf

# from the extension you know this is a 2-port network
ring_slot_sim = rf.Network('ring slot array simulation.s2p')
ring_slot_meas = rf.Network('ring slot array measured.s1p')

pylab.figure(1)
pylab.title('WR-10 Ringslot Array Simulated vs Measured')
# if no indecies are passed to the plot command it will plot all
# available s-parameters
ring_slot_sim.plot_s_db(0,0, label='Simulated')
ring_slot_meas.plot_s_db(0,0, label='Measured')
pylab.show()
```

1.3.6 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 Calibration

Contents

- Calibration
 - Intro
 - One-Port
 - Two-port
 - * A note on switch-terms
 - Simple Two Port
 - * Using s1p ideals in two-port calibration

1.4.1 Intro

This page describes 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 corresponding set ideal responses.

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

Calibrations are performed through a Calibration class, which makes creating and working with calibrations easy. Since skrf-1.2 the Calibration class only requires two pieces of information:

- a list of measured Networks
- a list of ideal Networks

The Network elements in each list must all be similar, (same #ports, same 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 for explicitness, such as,

- calibration type
- frequency information
- reciprocity of embedding networks
- etc

When this information is not provided skrf will determine it through inspection.

1.4.2 One-Port

See `example_oneport_calibration` for examples.

Below are (hopefully) self-explanatory examples of increasing complexity, which should illustrate, by example, how to make a calibration. Simple One-port

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

```
import skrf as rf

## created necessary data for Calibration class

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

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

## create a Calibration instance
cal = rf.Calibration(\
```

```

        ideals = my_ideals,
        measured = my_measured,
    )

## run, and apply calibration to a DUT

# run calibration algorithm
cal.run()

# apply it to a dut
dut = rf.Network('my_dut.s1p')
dut_calcd = cal.apply_cal(dut)

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

```

Concise One-port

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

```

1.4.3 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).

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>

A note on switch-terms

Switch-terms are explained in a paper by Roger Marks³. Basically, switch-terms account for the fact that the error networks change slightly depending on which port is being excited. This is due to the hardware of the VNA.

So how do you measure switch terms? With a custom measurement configuration on the VNA itself. `mwavpey` 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 your VNA.

See `example_twoport_calibration` for and `example`

1.4.4 Simple Two Port

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

```
import skrf as rf

## created necessary data for Calibration class

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

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

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

## run, and apply calibration to a DUT

# run calibration algorithm
cal.run()

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

# plot results
```

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>

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

Using s1p 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

1.5 Circuit Design

Contents

- Circuit Design
 - Intro
 - Media's Supported by skrf
 - Creating Individual Networks
 - Building Cicuits
 - Single Stub Tuner
 - Optimizing Designs

1.5.1 Intro

skrf has basic support for microwave circuit design. Network synthesis is accomplished through the Media Class (`skrf.media`), which represent a transmission line object for a given medium. A Media object contains properties such as propagation constant and characteristic impedance, that are needed to generate network components.

Typically circuit design is done within a given frequency band. Therefore every Media object is created with a Frequency object to relieve the user of repetitously providing frequency information for each new network created.

1.5.2 Media's Supported by skrf

Below is a list of mediums types supported by skrf,

- DistributedCircuit
- Freespace
- RectangularWaveguide
- CPW

More info on all of these classes can be found in the media sub-module section of `skrf.media` mavepy's API.

Here is an example of how to initialize a Media object representing a freespace from 10-20GHz:

```
import skrf as rf
freq = rf.Frequency(10,20,101,'ghz')
my_media = rf.media.Freespace(freq)
```

Here is another example constructing a coplanar waveguide media. The instance has a 10um center conductor and gap of 5um, on a substrate with relative permativity of 10.6,:

```
freq = rf.Frequency(500,750,101,'ghz')
my_media = rf.media.CPW(freq, w=10e-6, s=5e-6, ep_r=10.6)
```

or a WR10 Rectangular Waveguide:

```
from scipy.constants import * # for the 'mil' unit
freq = rf.Frequency(75,110,101,'ghz')
my_media = rf.media.RectangularWaveguide(freq, a=100*mil)
```

1.5.3 Creating Individual Networks

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

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

Details for each component and usage help can be found in their doc-strings. So `help(my_media.short)` should provide you with enough details to create a short-circuit component. To create a 1-port network for a short,

```
my_media.short()
```

to create a 90deg section of transmission line, with characteristic impedance of 30 ohms:

```
my_media.line(d=90,unit='deg',z0=30)
```

Network components specific to a given medium, such as `cpw_short`, or `microstrip_bend`, are implemented in by the Media Classes themselves.

1.5.4 Building Cicuits

Circuits can be built in an intuitive maner from individual networks. To build a the 90deg `delay_short` standard can be made by:

```
delay_short_90deg = my_media.line(90, 'deg') ** my_media.short()
```

For frequently used circuits, it may be worthwhile creating a function for something like this:

```
def delay_short(wb, *args, **kwargs):
    return my_media.line(*args, **kwargs) ** my_media.short()
```

```
delay_short(wb, 90, 'deg')
```

This is how many of skrf's network components are made internally.

To connect networks with more than two ports together, use the `connect()` function. You must provide the connect function with the two networks to be connected and the port indices (starting from 0) to be connected.

To connect port# '0' of ntwkA to port# '3' of ntwkB:

```
ntwkC = rf.connect(ntwkA, 0, ntwkB, 3)
```

Note that the connect function takes into account port impedances. To create a two-port network for a shunted delayed open, you can create an ideal 3-way splitter (a 'tee') and connect the delayed open to one of its ports, like so:

```
tee = my_media.tee()
delay_open = my_media.delay_open(40, 'deg')

shunt_open = connect(tee, 1, delay_open, 0)
```

1.5.5 Single Stub Tuner

This is an example of how to design a single stub tuning network to match a 100ohm resistor to a 50 ohm environment.

```
# calculate reflection coefficient off a 100ohm
Gamma0 = rf.zl_2_Gamma0(z0=50, z1=100)

# create the network for the 100ohm load
load = my_media.load(Gamma0)

# create the single stub network, parameterized by two delay lengths
# in units of 'deg'
single_stub = my_media.shunt_delay_open(120, 'deg') ** my_media.line(40, 'deg')

# the resulting network
result = single_stub ** load

result.plot_s_db()
```

1.5.6 Optimizing Designs

The abilities of scipy's optimizers can be used to automate network design. To automate the single stub design, we can create a 'cost' function which returns something we want to minimize, such as the reflection coefficient magnitude at band center.

```
from scipy.optimize import fmin

# the load we are trying to match
load = my_media.load(rf.zl_2_Gamma0(100))

# single stub generator function
```

```
def single_stub(wb,d0,d1):
    return my_media.shunt_open(d1,'deg')**my_media.line(d0,'deg')

# cost function we want to minimize (note: this uses sloppy namespace)
def cost(d):
    return (single_stub(wb,d[0],d[1]) ** load)[100].s_mag.squeeze()

# initial guess of optimal delay lengths in degrees
d0= 120,40 # initial guess

#determine the optimal delays
d_opt = fmin(cost,(120,40))
```

1.6 Uncertainty Estimation

Contents

- Uncertainty Estimation
 - Simple Case

scikit-rf can be used to calculate uncertainty estimates given a set of networks. The `NetworkSet` object holds sets of networks and provides automated methods for calculating and displaying uncertainty bounds.

Although the uncertainty estimation functions operate on any set of networks, the topic of uncertainty estimation is frequently associated with calibration uncertainty. That is, how certain can one be in results of a calibrated measurement.

1.6.1 Simple Case

Assume that numerous touchstone files, representing redundant measurements of a single network are made, such as:

```
In [24]: ls *ro*
ro,0.s1p ro,1.s1p ro,2.s1p ro,3.s1p ro,4.s1p ro,5.s1p
```

In case you are curious, the network in this example is an open rectangular waveguide radiating into freespace. The numerous touchstone files `ro,0.s1p`, `ro,1.s1p`, ... , are redundant measurements on which we would like to calculate the mean response, with uncertainty bounds.

This is most easily done by constructing `NetworkSet` object. The fastest way to achieve this is to use the convenience function `load_all_touchstones()`, which returns a dictionary with `Network` objects for values.

```
import pylab
import skrf as rf

ro_set = rf.NetworkSet(\
    rf.load_all_touchstones('.',contains='ro').values(),\
    name = 'Radiating Open')

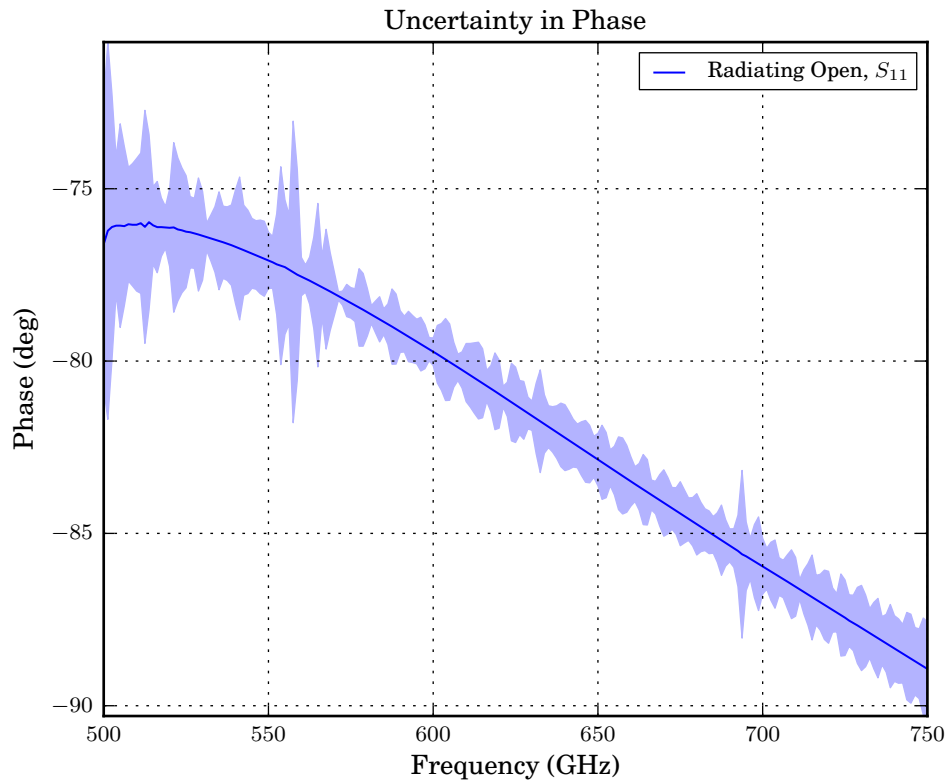
pylab.figure()
pylab.title('Uncertainty in Phase')
ro_set.plot_uncertainty_bounds_s_deg()

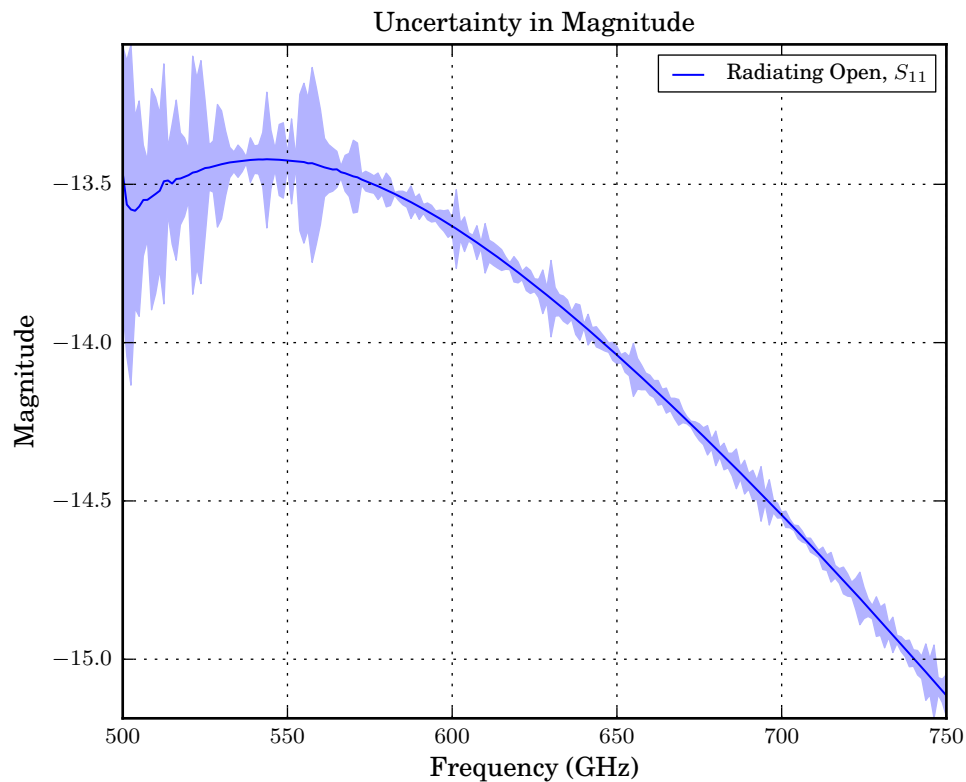
pylab.figure()
pylab.title('Uncertainty in Magnitude')
```



```
ro_set.plot_uncertainty_bounds_s_db()
```

```
pylab.show()
```





1.7 Developing skrf

Contents

- Developing skrf
 - Introduction
 - Contributing Code
 - Contribute Documentation
 - Creating Tests

1.7.1 Introduction

Welcome to the skrf's developer docs! This page is for those who are interested in participating those who are interested in developing scikit-rf.

Starting in February, 2012, skrf's codebase has been versioned using [git](#) , and hosted on github at <https://github.com/scikit-rf/scikit-rf>. The easiest way to contribute to any part of scikit-rf is to create an account on github, but if you are not familiar with git, dont hesitate to contact me directly by email at arsenovic@virginia.edu.

1.7.2 Contributing Code

skrf uses the *Fork + Pull* collaborative development model. Please see github's page on this for more information <http://help.github.com/send-pull-requests/>

1.7.3 Contribute Documentation

skrf's documentation is generated using [sphinx](#). The documentation source code is written using reStructured Text, and can be found in `docs/sphinx/source/`. The reference documentation for the submodules, classes, and functions are documented following the [conventions](#) put forth by Numpy/Scipy. Improvements or new documentation is welcomed, and can be submitted using github as well.

1.7.4 Creating Tests

skrf employs the python module *unittest* for testing. The test's are located in `skrf/testCases/`.

EXAMPLES

2.1 One-Port Calibration

2.1.1 Instructive

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

```
import skrf as rf

## created necessary data for Calibration class

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

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

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

## run, and apply calibration to a DUT

# run calibration algorithm
cal.run()

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

# plot results
```

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

2.1.2 Concise

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

```
import skrf as rf

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

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

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

2.2 Two-Port Calibration

This is an example of how to setup two-port calibration. For more detailed explanation see calibration:

```
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()
```

2.3 VNA Noise Analysis

This example records a series of sweeps from a VNA to touchstone files, named in a chronological order. These are then used to characterize the noise of a VNA.

2.3.1 Touchstone File Retrieval

```
import skrf as rf
import os, datetime

nsweeps = 101 # number of sweeps to take
dir = datetime.datetime.now().date().__str__() # directory to save files in

myvna = rf.vna.HP8720() # HP8510 also available
os.mkdir(dir)
for k in range(nsweeps):
    print k
    ntwk = myvna.s11
    date_string = datetime.datetime.now().__str__().replace(':', '-')
    ntwk.write_touchstone(dir + '/' + date_string)

myvna.close()
```

2.3.2 Noise Analysis

Calculates and plots various metrics of noise, given a directory of touchstone files, as would be created from the previous script.

```
import skrf as rf
from pylab import *

dir = '2010-12-03' # directory of touchstone files
npoints = 3 # number of frequency points to calculate statistics for

# load all touchstones in directory into a dictionary, and sort keys
data = rf.load_all_touchstones(dir+'/')
keys=data.keys()
keys.sort()

# length of frequency vector of each network
```

```
f_len = data[keys[0]].frequency.npoints
# frequency vector indecies at which we will calculate the statistics
f_vector = [int(k) for k in linspace(0,f_len-1, npoints)]

#loop through the frequencies of interest and calculate statistics
for f in f_vector:
    # for legends
    f_scaled = data[keys[0]].frequency.f_scaled[f]
    f_unit = data[keys[0]].frequency.unit

    # z is 1d complex array of the s11 at the current frequency, it is
    # as long as the number of touchstone files
    z = array( [(data[keys[k]]).s[f,0,0] for k in range(len(keys))])
    phase_change = rf.complex_2_degree(z * 1/z[0])
    phase_change = phase_change - mean(phase_change)
    mag_change = rf.complex_2_magnitude(z-z[0])

    figure(1)
    title('Complex Drift')
    plot(z.real,z.imag,'.',label='f = %i%s' % (f_scaled,f_unit))
    axis('equal')
    legend()
    rf.smith()

    figure(2)
    title('Phase Drift vs. Time')
    xlabel('Sample [n]')
    ylabel('Phase From Mean [deg]')
    plot(phase_change,label='f = %i%s, $\sigma$=%.1f$' % (f_scaled,f_unit,std(phase_change)))
    legend()

    figure(3)
    title('Phase Drift Distribution')
    xlabel('Phase From Mean[deg]')
    ylabel('Frequency Of Occurrence')
    hist(phase_change,alpha=.5,bins=21,histtype='stepfilled',\
         label='f = %i%s, $\sigma$=%.1f$' % (f_scaled,f_unit,std(phase_change)))
    legend()
    figure(4)
    title('FFT of Phase Drift')
    ylabel('Power [dB]')
    xlabel('Sample Frequency [?]' )
    plot(log10(abs(fftshift(fft(phase_change))))[len(keys)/2+1:])

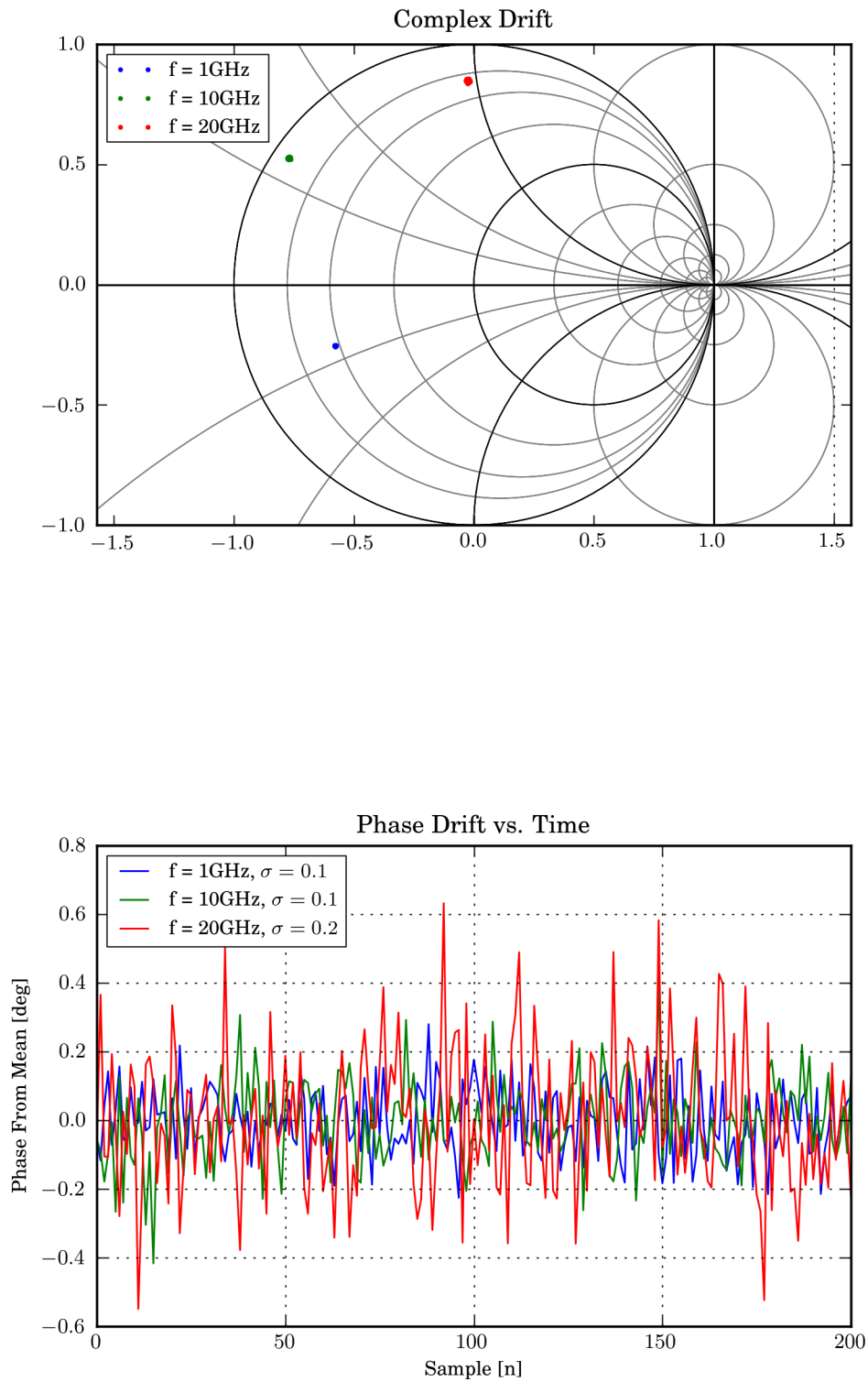
draw();show();
```

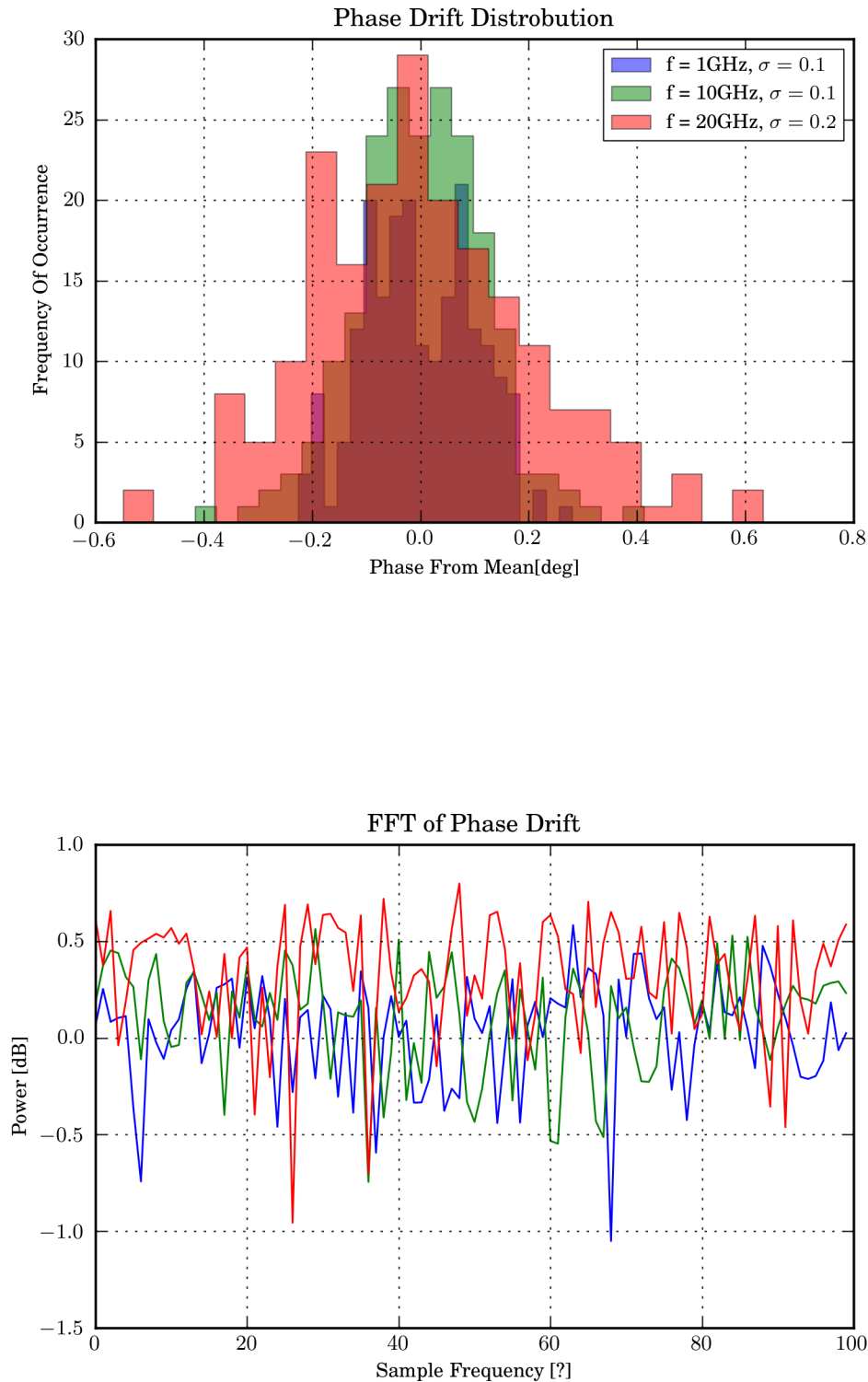
2.4 Circuit Design: Single Stub Matching Network

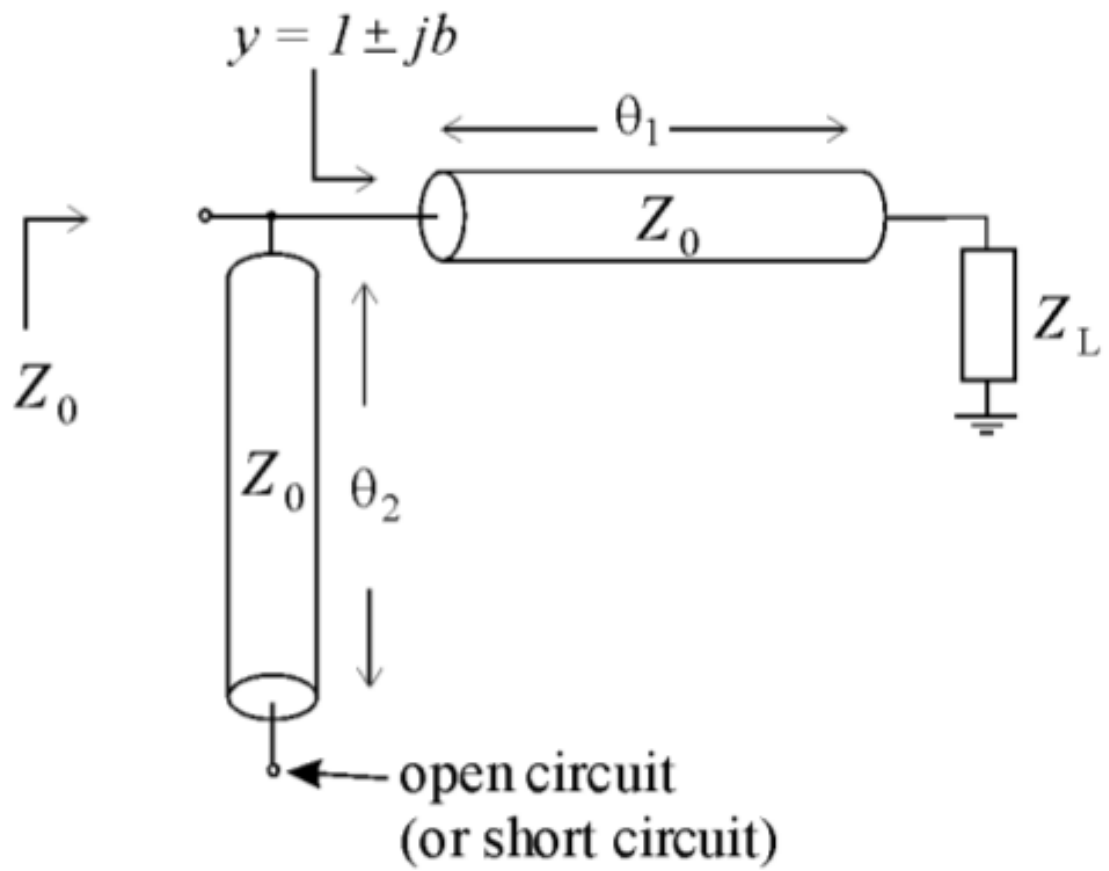
2.4.1 Introduction

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, 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 return loss magnitude off the matched load, vs series and shunt line lengths. The optimal designs are then seen as the minima of a 2D surface.

2.4.2 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 = 51      # 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
    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] )

# show the resultant return loss for the parameters space
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')

from mpl_toolkits.mplot3d import Axes3D

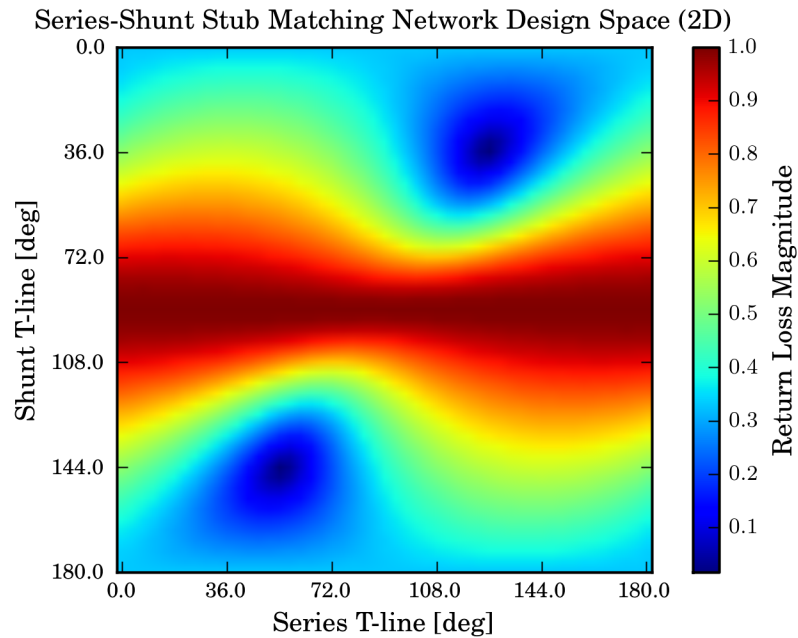
fig=figure()
ax = Axes3D(fig)
x,y = meshgrid(d_range, d_range)
ax.plot_surface(x,y,output, rstride=1, cstride=1,cmap=cm.jet)
```

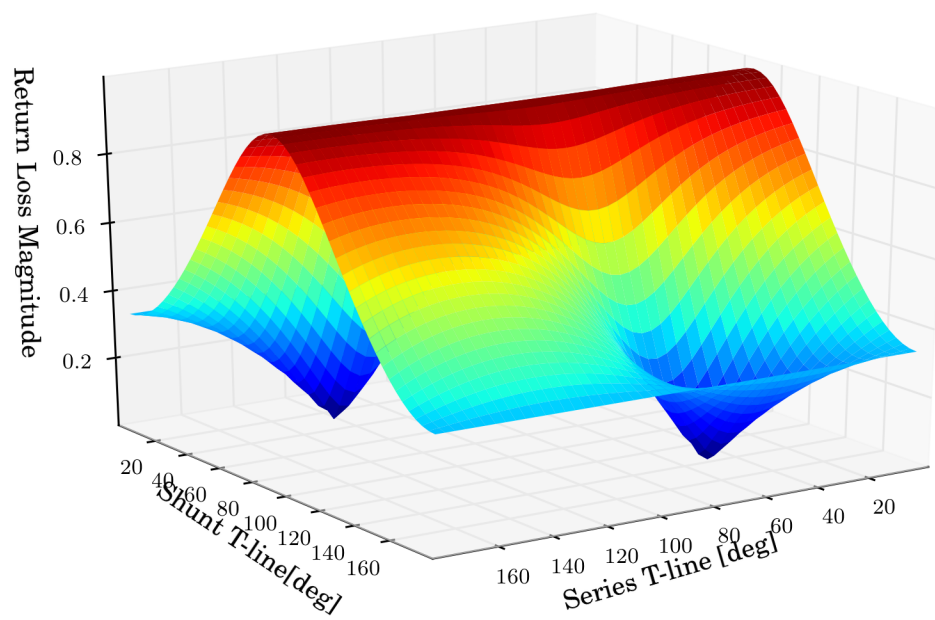
```

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)')
draw()
show()

```

2.4.3 Output





REFERENCE

3.1 Major Classes

- `Network`
- `NetworkSet`
- `Frequency`
- `Calibration`

3.2 Modules

3.2.1 `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.

Network Class

`Network([touchstone_file, name])` A n-port electrical network.

`skrf.network.Network`

class `skrf.network.Network` (*touchstone_file=None, name=None*)
A n-port electrical network.

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

- scattering parameter matrix (s-matrix)
- port characteristic impedance matrix
- frequency information

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

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

Individual components of the s-matrix are accesable through properties as well. These also return `numpy.ndarray`'s.

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

The following `Network` operators are available:

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

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

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

Generally, `Network` objects are created from touchstone files upon initialization (see `__init__()`), or are created from a `Media` object. `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

Attributes

<code>f</code>	the frequency vector for the network, in Hz.
<code>frequency</code>	frequency information for the network.
<code>inv</code>	a <code>Network</code> object with 'inverse' s-parameters.
<code>nports</code>	the number of ports the network has.
<code>number_of_ports</code>	the number of ports the network has.
<code>passivity</code>	passivity metric for a multi-port network.
<code>s</code>	the scattering parameter matrix [#] .
<code>t</code>	t-parameters, aka scattering transfer parameters [#] .
<code>y</code>	admittance parameters
<code>z</code>	impedance parameters
Continued on next page	

Table 3.2 – continued from previous page

<code>z0</code>	the characteristic impedance[s] of the network ports.
-----------------	---

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

the frequency vector for the network, in Hz.

Returns `f` : `numpy.ndarray`

frequency vector in Hz

See Also:`frequency` frequency property that holds all frequency information**skrf.network.Network.frequency**`Network.frequency`

frequency information for the network.

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

Returns `frequency` : `Frequency` object

frequency information for the network.

See Also:`f` property holding frequency vector in Hz`change_frequency` updates frequency property, and interpolates s-parameters if needed`interpolate` interpolate function based on new frequency info**skrf.network.Network.inv**`Network.inv`a `Network` object with ‘inverse’ s-parameters.

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

Returns `inv` : a `Network` objecta `Network` object with ‘inverse’ s-parameters.**See Also:**`inv` function which implements the inverse s-matrix**skrf.network.Network.nports**`Network.nports`

the number of ports the network has.

Returns `number_of_ports` : number

the number of ports the network has.

skrf.network.Network.number_of_ports**Network.number_of_ports**

the number of ports the network has.

Returns **number_of_ports** : number

the number of ports the network has.

skrf.network.Network.passivity**Network.passivity**

passivity metric for a multi-port network.

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

mathematically, this is a test for unitary-ness of the s-parameter matrix ¹.

for two port this is

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

in general it is

$$S^H \cdot S$$

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

Returns **passivity** : numpy.ndarray of shape $fxn \times n$ **References****skrf.network.Network.s****Network.s**the scattering parameter matrix ².

s-matrix is a 3 dimensional numpy.ndarray which has shape $fxn \times n$, where f is frequency axis and n is number of ports. Note that indexing starts at 0!, so s_{11} can be accessed by taking the slice $s[:,0,0]$.

Returns **s** : complex numpy.ndarry of shape $fxn \times n$

the scattering parameter matrix.

References**skrf.network.Network.t****Network.t**t-parameters, aka scattering transfer parameters ³

this is also known as the wave cascading matrix, and is only defined for a 2-port Network

Returns **t** : complex numpy.ndarry of shape $fxn \times n$

t-parameters, aka scattering transfer parameters

¹ http://en.wikipedia.org/wiki/Scattering_parameters#Lossless_networks

² http://en.wikipedia.org/wiki/Scattering_parameters

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

References

skrf.network.Network.y

`Network.y`
admittance parameters

skrf.network.Network.z

`Network.z`
impedance parameters

skrf.network.Network.z0

`Network.z0`
the 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, that is a function of frequency, `z0` is stored internally as a *fxn* array.

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

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

Methods

<code>__init__</code>	constructor.
<code>add_noise_polar</code>	adds a complex zero-mean gaussian white-noise.
<code>add_noise_polar_flatband</code>	adds a flatband complex zero-mean gaussian white-noise signal of
<code>copy</code>	returns a copy of this Network
<code>flip</code>	swaps the ports of a two port Network
<code>interpolate</code>	calculates an interpolated network.
<code>interpolate_self</code>	interpolates s-parameters given a new
<code>interpolate_self_npoints</code>	interpolate network based on a new number of frequency points
<code>multiply_noise</code>	multiplies a complex bivariate gaussian white-noise signal
<code>nudge</code>	perturb s-parameters by small amount. this is useful to
<code>plot_it_all</code>	
<code>plot_passivity</code>	plots the passivity of a network, possibly for a specific port.
<code>plot_s_smith</code>	plots the scattering parameter on a smith chart
<code>read_touchstone</code>	loads values from a touchstone file.
<code>renumber</code>	renumbers some ports of a two port Network
<code>write_touchstone</code>	write a contents of the <code>Network</code> to a touchstone file.

skrf.network.Network.__init__

`Network.__init__` (*touchstone_file=None, name=None*)
constructor.

Constructs a Network, and optionally populates the s-matrix and frequency information from touchstone file.

Parameters `file`: string :

if given will load information from touchstone file, optional

name: string :

name of this network, optional

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

skrf.network.Network.flip

`Network.flip()`

swaps the ports of a two port Network

skrf.network.Network.interpolate

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

calculates an interpolated network.

The default interpolation type is linear. see Notes for how to use other interpolation types.

Parameters `new_frequency` : `Frequency`

frequency information to interpolate at

****kwargs** : keyword arguments

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

Returns `result` : `Network`

an interpolated Network

Notes

useful keyword for `scipy.interpolate.interp1d()`,

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.

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

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

The default interpolation type is linear. see Notes for how to use other interpolation types.

Parameters `new_frequency` : `Frequency`

frequency information to interpolate at

****kwargs** : keyword argumentspassed to `scipy.interpolate.interp1d()` initializer.**See Also:**`interpolate` same function, but returns a new Network**Notes**useful keyword for `scipy.interpolate.interp1d()`,**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.**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 argumentspassed 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.**skrf.network.Network.multiply_noise**`Network.multiply_noise` (*mag_dev*, *phase_dev*, ***kwargs*)

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

takes: `mag_dev`: standard deviation of magnitude `phase_dev`: standard deviation of phase [in degrees] `n_ports`: number of ports. default to 1**returns:** nothing

skrf.network.Network.nudge

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

perturb s-parameters by small amount. this is useful to work-around numerical bugs.

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

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_smith

`Network.plot_s_smith` (*m=None, n=None, r=1, ax=None, show_legend=True, chart_type='z', *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

***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.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** : string

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 :

skrf.network.Network.write_touchstone

`Network.write_touchstone(filename=None, dir='./')`

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 './'.

Notes

format supported at the moment is, HZ S RI

The functionality of this function should take place in the `touchstone` class.

Connecting Networks

<code>connect(ntwkA, k, ntwkB, l)</code>	connect two n-port networks together.
<code>innerconnect(ntwkA, k, l)</code>	connect two ports of a single n-port network.
<code>cascade(ntwkA, ntwkB)</code>	cascade two 2-port Networks together
<code>de_embed(ntwkA, ntwkB)</code>	de-embed <i>ntwkA</i> from <i>ntwkB</i> . this calls <i>ntwkA.inv**ntwkB</i> .

`skrf.network.connect`

`skrf.network.connect (ntwkA, k, ntwkB, l)`

connect two n-port networks together.

specifically, connect port *k* on *ntwkA* to port *l* on *ntwkB*. The resultant network has (ntwkA.nports+ntwkB.nports-2) ports. The port index's ('k','l') start from 0. Port impedances **are** taken into account.

Parameters `ntwkA` : `Network`

network 'A'

`k` : int

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

`ntwkB` : `Network`

network 'B'

`l` : int

port index on *ntwkB*

Returns `ntwkC` : `Network`

new network of rank (ntwkA.nports + ntwkB.nports - 2*n)-ports

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*)
connect two 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

port indices on *ntwkA* (port indices start from 0)

Returns *ntwkC* : *Network*

new network of rank (ntwkA.nports+ntwkB.nports -2)-ports

See Also:

`connect_s` actual S-parameter connection algorithm.

`innerconnect_s` actual S-parameter connection algorithm.

Notes

a 2-port 'mismatch' network between the two connected ports.

Examples

To connect ports '0' and port '1' on *ntwkA*

```
>>> ntwkA = rf.Network('ntwkA.s3p')
>>> ntwkC = rf.innerconnect(ntwkA, 0, 1)
```

skrf.network.cascade

`skrf.network.cascade` (*ntwkA*, *ntwkB*)
cascade two 2-port Networks together

connects port 1 of *ntwkA* to port 0 of *ntwkB*. This calls `connect(ntwkA,1, ntwkB,0)`, which is a more general function.

Parameters *ntwkA* : *Network*

network *ntwkA*

ntwkB : *Network*

network *ntwkB*

Returns *C* : *Network*

the resultant network of `ntwkA` cascaded with `ntwkB`

See Also:

`connect` connects two Networks together at arbitrary ports.

`skrf.network.de_embed`

`skrf.network.de_embed(ntwkA, ntwkB)`

de-embed `ntwkA` from `ntwkB`. this calls `ntwkA.inv**ntwkB`. the syntax of cascading an inverse is more explicit, it is recommended that it be used instead of this function.

Parameters `ntwkA` : `Network`

network `ntwkA`

`ntwkB` : `Network`

network `ntwkB`

Returns `C` : `Network`

the resultant network of `ntwkB` de-embedded from `ntwkA`

See Also:

`connect` connects two Networks together at arbitrary ports.

Interpolation

<code>Network.interpolate(new_frequency, **kwargs)</code>	calculates an interpolated network.
<code>Network.interpolate_self(new_frequency, **kwargs)</code>	interpolates s-parameters given a new
<code>Network.interpolate_self_npoints(npoints, ...)</code>	interpolate network based on a new number of frequency points

`skrf.network.Network.interpolate`

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

calculates an interpolated network.

The default interpolation type is linear. see Notes for how to use other interpolation types.

Parameters `new_frequency` : `Frequency`

frequency information to interpolate at

****kwargs** : keyword arguments

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

Returns `result` : `Network`

an interpolated Network

Notes

useful keyword for `scipy.interpolate.interpld()`,

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.

skrf.network.Network.interpolate_self

`Network.interpolate_self(new_frequency, **kwargs)`
interpolates s-parameters given a new :class:`~skrf.frequency.Frequency` object.

The default interpolation type is linear. see Notes for how to use other interpolation types.

Parameters `new_frequency` : `Frequency`

frequency information to interpolate at

****kwargs** : keyword arguments

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

See Also:

`interpolate` same function, but returns a new Network

Notes

useful keyword for `scipy.interpolate.interpld()`,

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.

skrf.network.Network.interpolate_self_npoints

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

Parameters `npoints` : int

number of frequency points

****kwargs** : keyword arguments

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

See Also:

`interpolate_self` same functionality but takes a Frequency object

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

Supporting Functions

<code>inv(s)</code>	Calculates 'inverse' s-parameter matrix, used for de-embedding
<code>connect_s(A, k, B, l)</code>	connect two n-port networks' s-matrices together.
<code>innerconnect_s(A, k, l)</code>	connect two ports of a single n-port network's s-matrix.
<code>s2z(s[, z0])</code>	convert scattering parameters to impedance parameters [#]
Continued on next page	

Table 3.6 – continued from previous page

<code>s2y(s[, z0])</code>	convert scattering parameters to admittance parameters [#]
<code>s2t(s)</code>	converts scattering parameters to scattering transfer parameters.
<code>z2s(z[, z0])</code>	convert impedance parameters to scattering parameters [#]
<code>z2y(z)</code>	convert impedance parameters to admittance parameters [#]
<code>z2t(z)</code>	Not Implemented yet
<code>y2s(y[, z0])</code>	convert admittance parameters to scattering parameters [#]
<code>y2z(y)</code>	convert admittance parameters to impedance parameters [#]
<code>y2t(y)</code>	Not Implemented Yet
<code>t2s(t)</code>	converts scattering transfer parameters to scattering parameters
<code>t2z(t)</code>	Not Implemented Yet
<code>t2y(t)</code>	Not Implemented Yet

`skrf.network.inv`

`skrf.network.inv(s)`

Calculates ‘inverse’ s-parameter matrix, used for de-embedding

This is not literally the inverse of the s-parameter matrix. Instead, it is defined such that the inverse of the s-matrix cascaded with itself is unity.

$$inv(s) = t2s(s2t(s)^{-1})$$

where x^{-1} is the matrix inverse. In words, this is the inverse of the scattering transfer parameters matrix transformed into a scattering parameters matrix.

Parameters `s` : numpy.ndarray (shape fx2x2)

scattering parameter matrix.

Returns `s'` : numpy.ndarray

inverse scattering parameter matrix.

See Also:

`t2s` converts scattering transfer parameters to scattering parameters

`s2t` converts scattering parameters to scattering transfer parameters

`skrf.network.connect_s`

`skrf.network.connect_s(A, k, B, l)`

connect two n-port networks’ s-matrices together.

specifically, connect port k on network A to port l on network B . The resultant network has $nports = (A.rank + B.rank - 2)$. This function operates on, and returns s-matrices. The function `connect()` operates on `Network` types.

Parameters `A` : numpy.ndarray

S-parameter matrix of A , shape is $fxn \times nxn$

`k` : int

port index on A (port indices start from 0)

`B` : numpy.ndarray

S-parameter matrix of B , shape is $fxn \times nxn$

l : int

port index on B

Returns **C** : numpy.ndarray

new S-parameter matrix

See Also:

`connect` operates on `Network` types

`innerconnect_s` function which implements the connection connection algorithm

Notes

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

skrf.network.innerconnect_s

`skrf.network.innerconnect_s(A, k, l)`

connect two ports of a single n-port network's s-matrix.

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

Parameters **A** : numpy.ndarray

S-parameter matrix of A , shape is $fxn \times nxn$

k : int

port index on A (port indices start from 0)

l : int

port index on A

Returns **C** : numpy.ndarray

new S-parameter matrix

Notes

The algorithm used to calculate the resultant network is called a 'sub-network growth', can be found in ⁴. The original paper describing the algorithm is given in ⁵.

⁴ 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>

⁵ 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 to impedance parameters ⁶

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

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 converts scattering parameters to impedance parameters

s2y converts scattering parameters to admittance parameters

s2t converts scattering parameters to scattering transfer parameters

z2s converts impedance parameters to scattering parameters

z2y converts impedance parameters to admittance parameters

z2t converts impedance parameters to scattering transfer parameters

y2s converts admittance parameters to impedance parameters

y2z converts admittance parameters to admittance parameters

y2t converts admittance parameters to scattering transfer parameters

t2s converts scattering transfer parameters to scattering parameters

t2z converts scattering transfer parameters to impedance parameters

t2y converts scattering transfer parameters to admittance parameters

References

skrf.network.s2y

`skrf.network.s2y` (*s*, *z0*=50)
convert scattering parameters to admittance parameters ⁷

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

Parameters *s* : complex array-like

⁶ http://en.wikipedia.org/wiki/impedance_parameters

⁷ http://en.wikipedia.org/wiki/Admittance_parameters

scattering parameters

z0 : complex array-like or number

port impedances

Returns **y** : complex array-like

admittance parameters

See Also:

s2z converts scattering parameters to impedance parameters

s2y converts scattering parameters to admittance parameters

s2t converts scattering parameters to scattering transfer parameters

z2s converts impedance parameters to scattering parameters

z2y converts impedance parameters to admittance parameters

z2t converts impedance parameters to scattering transfer parameters

y2s converts admittance parameters to impedance parameters

y2z converts admittance parameters to admittance parameters

y2t converts admittance parameters to scattering transfer parameters

t2s converts scattering transfer parameters to scattering parameters

t2z converts scattering transfer parameters to impedance parameters

t2y converts scattering transfer parameters to admittance parameters

References

skrf.network.s2t

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

converts scattering parameters to scattering transfer parameters.

transfer parameters⁸ are also referred to as ‘wave cascading matrix’, this function only operates on 2-port networks.

Parameters **s** : numpy.ndarray (shape *fx2x2*)

scattering parameter matrix

Returns **t** : numpy.ndarray

scattering transfer parameters (aka wave cascading matrix)

See Also:

t2s converts scattering transfer parameters to scattering parameters

inv calculates inverse s-parameters

⁸ http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters

References

skrf.network.z2s

`skrf.network.z2s(z, z0=50)`
convert impedance parameters to scattering parameters ⁹

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

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 converts scattering parameters to impedance parameters
s2y converts scattering parameters to admittance parameters
s2t converts scattering parameters to scattering transfer parameters
z2s converts impedance parameters to scattering parameters
z2y converts impedance parameters to admittance parameters
z2t converts impedance parameters to scattering transfer parameters
y2s converts admittance parameters to impedance parameters
y2z converts admittance parameters to impedance parameters
y2t converts admittance parameters to scattering transfer parameters
t2s converts scattering transfer parameters to scattering parameters
t2z converts scattering transfer parameters to impedance parameters
t2y converts scattering transfer parameters to admittance parameters

References

skrf.network.z2y

`skrf.network.z2y(z)`
convert impedance parameters to admittance parameters ¹⁰

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

Parameters **z** : complex array-like

⁹ http://en.wikipedia.org/wiki/impedance_parameters

¹⁰ http://en.wikipedia.org/wiki/Two-port_network

impedance parameters

Returns `y` : complex array-like

admittance parameters

See Also:

s2z converts scattering parameters to impedance parameters

s2y converts scattering parameters to admittance parameters

s2t converts scattering parameters to scattering transfer parameters

z2s converts impedance parameters to scattering parameters

z2y converts impedance parameters to admittance parameters

z2t converts impedance parameters to scattering transfer parameters

y2s converts admittance parameters to impedance parameters

y2z converts admittance parameters to admittance parameters

y2t converts admittance parameters to scattering transfer parameters

t2s converts scattering transfer parameters to scattering parameters

t2z converts scattering transfer parameters to impedance parameters

t2y converts scattering transfer parameters to admittance parameters

References

skrf.network.z2t

`skrf.network.z2t(z)`

Not Implemented yet

convert impedance parameters to scattering transfer parameters ¹¹

Parameters `z` : complex array-like or number

impedance parameters

Returns `s` : complex array-like or number

scattering parameters

See Also:

s2z converts scattering parameters to impedance parameters

s2y converts scattering parameters to admittance parameters

s2t converts scattering parameters to scattering transfer parameters

z2s converts impedance parameters to scattering parameters

z2y converts impedance parameters to admittance parameters

z2t converts impedance parameters to scattering transfer parameters

y2s converts admittance parameters to impedance parameters

¹¹ http://en.wikipedia.org/wiki/Two-port_network

y2z converts admittance parameters to impedance parameters

y2z converts admittance parameters to scattering transfer parameters

t2s converts scattering transfer parameters to scattering parameters

t2z converts scattering transfer parameters to impedance parameters

t2y converts scattering transfer parameters to admittance parameters

References

skrf.network.y2s

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

convert admittance parameters to scattering parameters ¹²

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

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 converts scattering parameters to impedance parameters

s2y converts scattering parameters to admittance parameters

s2t converts scattering parameters to scattering transfer parameters

z2s converts impedance parameters to scattering parameters

z2y converts impedance parameters to impedance parameters

z2t converts impedance parameters to scattering transfer parameters

y2s converts admittance parameters to impedance parameters

y2z converts admittance parameters to impedance parameters

y2z converts admittance parameters to impedance parameters

t2s converts scattering transfer parameters to scattering parameters

t2z converts scattering transfer parameters to impedance parameters

t2y converts scattering transfer parameters to admittance parameters

¹² http://en.wikipedia.org/wiki/Admittance_parameters

References

skrf.network.y2z

skrf.network.**y2z**(y)
convert admittance parameters to impedance parameters ¹³

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

Parameters y : complex array-like

admittance parameters

Returns z : complex array-like

impedance parameters

See Also:

s2z converts scattering parameters to impedance parameters

s2y converts scattering parameters to admittance parameters

s2t converts scattering parameters to scattering transfer parameters

z2s converts impedance parameters to scattering parameters

z2y converts impedance parameters to admittance parameters

z2t converts impedance parameters to scattering transfer parameters

y2s converts admittance parameters to scattering parameters

y2z converts admittance parameters to impedance parameters

y2t converts admittance parameters to scattering transfer parameters

t2s converts scattering transfer parameters to scattering parameters

t2z converts scattering transfer parameters to impedance parameters

t2y converts scattering transfer parameters to admittance parameters

References

skrf.network.y2t

skrf.network.**y2t**(y)
Not Implemented Yet
convert admittance parameters to scattering-transfer parameters ¹⁴

Parameters y : complex array-like or number

impedance parameters

Returns t : complex array-like or number

scattering parameters

¹³ http://en.wikipedia.org/wiki/Admittance_parameters

¹⁴ http://en.wikipedia.org/wiki/Two-port_network

See Also:

s2z converts scattering parameters to impedance parameters
s2y converts scattering parameters to admittance parameters
s2t converts scattering parameters to scattering transfer parameters
z2s converts impedance parameters to scattering parameters
z2y converts impedance parameters to impedance parameters
z2t converts impedance parameters to scattering transfer parameters
y2s converts admittance parameters to impedance parameters
y2z converts admittance parameters to impedance parameters
y2t converts admittance parameters to scattering transfer parameters
t2s converts scattering transfer parameters to scattering parameters
t2z converts scattering transfer parameters to impedance parameters
t2y converts scattering transfer parameters to admittance parameters

References**skrf.network.t2s**

`skrf.network.t2s(t)`

converts scattering transfer parameters to scattering parameters

transfer parameters ¹⁵ are also referred to as ‘wave cascading matrix’, this function only operates on 2-port networks. this function only operates on 2-port scattering parameters.

Parameters **t** : numpy.ndarray (shape fx2x2)

scattering transfer parameters

Returns **s** : numpy.ndarray

scattering parameter matrix.

See Also:

t2s converts scattering transfer parameters to scattering parameters
inv calculates inverse s-parameters

References**skrf.network.t2z**

`skrf.network.t2z(t)`

Not Implemented Yet

Convert scattering transfer parameters to impedance parameters ¹⁶

¹⁵ http://en.wikipedia.org/wiki/Scattering_transfer_parameters#Scattering_transfer_parameters

¹⁶ http://en.wikipedia.org/wiki/Two-port_network

Parameters **t** : complex array-like or number

impedance parameters

Returns **z** : complex array-like or number

scattering parameters

See Also:

s2z converts scattering parameters to impedance parameters

s2y converts scattering parameters to admittance parameters

s2t converts scattering parameters to scattering transfer parameters

z2s converts impedance parameters to scattering parameters

z2y converts impedance parameters to impedance parameters

z2t converts impedance parameters to scattering transfer parameters

y2s converts admittance parameters to impedance parameters

y2z converts admittance parameters to impedance parameters

y2t converts admittance parameters to scattering transfer parameters

t2s converts scattering transfer parameters to scattering parameters

t2z converts scattering transfer parameters to impedance parameters

t2y converts scattering transfer parameters to admittance parameters

References

skrf.network.t2y

skrf.network.**t2y**(*t*)

Not Implemented Yet

Convert scattering transfer parameters to admittance parameters ¹⁷

Parameters **t** : complex array-like or number

t-parameters

Returns **y** : complex array-like or number

admittance parameters

See Also:

s2z converts scattering parameters to impedance parameters

s2y converts scattering parameters to admittance parameters

s2t converts scattering parameters to scattering transfer parameters

z2s converts impedance parameters to scattering parameters

z2y converts impedance parameters to impedance parameters

z2t converts impedance parameters to scattering transfer parameters

¹⁷ http://en.wikipedia.org/wiki/Two-port_network

y2s converts admittance parameters to impedance parameters
y2z converts admittance parameters to impedance parameters
y2z converts admittance parameters to scattering transfer parameters
t2s converts scattering transfer parameters to scattering parameters
t2z converts scattering transfer parameters to impedance parameters
t2y converts scattering transfer parameters to admittance parameters

References

Misc Functions

<code>average(list_of_networks)</code>	calculates the average network from a list of Networks.
<code>one_port_2_two_port(ntwk)</code>	calculates the two-port network given a symmetric, reciprocal and
<code>impedance_mismatch(z1, z2)</code>	creates a two-port s-matrix for a impedance mis-match
<code>load_all_touchstones([dir, contains, f_unit])</code>	loads all touchstone files in a given dir into a dictionary.
<code>write_dict_of_networks(ntwkDict[, dir])</code>	saves a dictionary of networks touchstone files in a given directory
<code>csv_2_touchstone(filename)</code>	converts a csv file to a <code>Network</code>

`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 :

a list of `Network` objects

Returns `ntwk` : `Network`

the resultant averaged Network

Notes

This same function can be accomplished with properties of a `NetworkSet` class.

Examples

```
>>> ntwk_list = [rf.Network('myntwk.slp'), rf.Network('myntwk2.slp')]  
>>> mean_ntwk = rf.average(ntwk_list)
```

`skrf.network.one_port_2_two_port`

`skrf.network.one_port_2_two_port` (*ntwk*)
calculates the two-port network given a symmetric, reciprocal and lossless one-port network.
takes: `ntwk`: a symmetric, reciprocal and lossless one-port network.

returns: `ntwk`: the resultant two-port Network

`skrf.network.impedance_mismatch`

`skrf.network.impedance_mismatch(z1, z2)`
creates a two-port s-matrix for a impedance mis-match

Parameters `z1`: number or array-like
complex impedance of port 1

`z2`: number or array-like
complex impedance of port 2

Returns `s`: 2-port s-matrix for the impedance mis-match

`skrf.network.load_all_touchstones`

`skrf.network.load_all_touchstones(dir='.', contains=None, f_unit=None)`
loads all touchtone files in a given dir into a dictionary.

Parameters `dir`: string
the path

`contains`: string
a string the filenames must contain to be loaded.

`f_unit`: ['hz', 'mhz', 'ghz']
the frequency unit to assign all loaded networks. see
`frequency.Frequency.unit`.

Returns `ntwkDict`: a dictionary with keys equal to the file name (without
a suffix), and values equal to the corresponding `ntwk` types

Examples

```
>>> ntwk_dict = rf.load_all_touchstones('.', contains='20v')
```

`skrf.network.write_dict_of_networks`

`skrf.network.write_dict_of_networks(ntwkDict, dir='.')`
saves a dictionary of networks touchstone files in a given directory

The filenames assigned to the touchstone files are taken from the keys of the dictionary.

Parameters `ntwkDict`: dictionary
dictionary of `Network` objects

`dir`: string
directory to write touchstone file to

skrf.network.csv_2_touchstone

skrf.network.csv_2_touchstone(filename)

converts a csv file to a `Network`

specifically, this converts csv files saved from a Rohde Shcwarz ZVA-40, and possibly other network analyzers, into a `Network` object.

Parameters filename : string

name of file

Returns ntwk : `Network` object

the network representing data in the csv file

3.2.2 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.

NetworkSet Class

`NetworkSet(ntwk_set[, name])` A set of Networks.

skrf.networkSet.NetworkSet

class skrf.networkSet.**NetworkSet**(ntwk_set, name=None)

A set of Networks.

This class allows functions on sets of Networks, such as mean or standard deviation, to be calculated conveniently. The results are returned in `Network` objects, so that they may be plotted and saved in like `Network` objects.

This class also provides methods which can be used to plot uncertainty bounds for a set of `Network`.

The names of the `NetworkSet` properties are generated dynamically upon initialization, and thus documentation for individual properties and methods is not available. However, the properties do follow the convention:

```
>>> my_network_set.function_name_network_property_name
```

For example, the complex average (mean) `Network` for a `NetworkSet` is:

```
>>> my_network_set.mean_s
```

This accesses the property 's', for each element in the set, and **then** calculates the 'mean' of the resultant set. The order of operations is important.

Results are returned as `Network` objects, so they may be plotted or saved in the same way as for `Network` objects:


```
>>> my_network_set.mean_s.plot_s_mag()
>>> my_network_set.mean_s.write_touchstone('mean_response')
```

If you are calculating functions that return scalar variables, then the result is accessible through the Network property `.s_re`. For example:

```
>>> std_s_deg = my_network_set.std_s_deg
```

This result would be plotted by:

```
>>> std_s_deg.plot_s_re()
```

The operators, properties, and methods of NetworkSet object are dynamically generated by private methods

- `__add_a_operator()`
- `__add_a_func_on_property()`
- `__add_a_element_wise_method()`
- `__add_a_plot_uncertainty()`

thus, documentation on the individual methods and properties are not available.

Attributes

<code>inv</code>	
<code>mean_s_db</code>	the mean magnitude in dB.
<code>std_s_db</code>	the mean magnitude in dB.

`skrf.networkSet.NetworkSet.inv`

`NetworkSet.inv`

`skrf.networkSet.NetworkSet.mean_s_db`

`NetworkSet.mean_s_db`

the mean magnitude in dB.

note:

the mean is taken on the magnitude before converted to db, so `magnitude_2_db(mean(s_mag))`

which is NOT the same as `mean(s_db)`

`skrf.networkSet.NetworkSet.std_s_db`

`NetworkSet.std_s_db`

the mean magnitude in dB.

note:

the mean is taken on the magnitude before converted to db, so `magnitude_2_db(mean(s_mag))`

which is NOT the same as `mean(s_db)`

Methods

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

skrf.networkSet.NetworkSet.__init__

NetworkSet.**__init__** (*ntwk_set*, *name=None*)

Initializer for NetworkSet

Parameters *ntwk_set* : list of `Network` objects

the set of `Network` objects

name : string

the name of the NetworkSet, given to the Networks returned from properties of this class.

skrf.networkSet.NetworkSet.copy

NetworkSet.**copy** ()

copies each network of the network set.

skrf.networkSet.NetworkSet.element_wise_method

NetworkSet.**element_wise_method** (*network_method_name*, **args*, ***kwargs*)

calls a given method of each element and returns the result as a new NetworkSet if the output is a Network.

skrf.networkSet.NetworkSet.from_zip

classmethod NetworkSet.**from_zip** (*zip_file_name*, *sort_filenames=True*, **args*, ***kwargs*)

creates a NetworkSet from a zipfile of touchstones.

Parameters *zip_file_name* : string

name of zipfile

sort_filenames: Boolean :

sort the filenames in teh zip file before constructing the NetworkSet

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

passed to NetworkSet constructor

Examples

```
>>>import skrf as rf >>>my_set = rf.NetworkSet.from_zip('myzip.zip')
```

skrf.networkSet.NetworkSet.plot_logsigma

`NetworkSet.plot_logsigma` (*label_axis=True, *args, **kwargs*)

plots the uncertainty for the set in units of log-sigma. Log-sigma is the complex standard deviation, plotted in units of dB's.

Parameters **args, **kwargs* : arguments

passed to `self.std_s.plot_s_db()`

skrf.networkSet.NetworkSet.plot_uncertainty_bounds_component

`NetworkSet.plot_uncertainty_bounds_component` (*attribute, m=0, n=0, type='shade', n_deviations=3, alpha=0.3, color_error=None, markevery_error=20, ax=None, ppf=None, kwargs_error={}, *args, **kwargs*)

plots mean value of the NetworkSet with +- uncertainty bounds in an Network's attribute. This is designed to represent uncertainty in a scalar component of the s-parameter. for example plotting the uncertainty in the magnitude would be expressed by,

`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. uncertainty 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 uncertainty bounds plot on smith chart.

This function plots the complex uncertainty of a NetworkSet as circles on the smith chart. At each frequency a circle with radii proportional to the complex standard deviation of the set at that frequency is drawn. Due to the fact that the *markersize* argument is in pixels, the radii can be scaled by the input argument *multiplier*.

default kwargs are { 'marker':'o', 'color':'b', 'mew':0, 'ls':'-', 'alpha':.1, 'label':None, }

Parameters *multiplier* : float

controls the circle sizes, by multiples of the standard deviation.

skrf.networkSet.NetworkSet.plot_uncertainty_bounds_s_db

`NetworkSet.plot_uncertainty_bounds_s_db(*args, **kwargs)`

this just calls `plot_uncertainty_bounds(attribute='s_mag',ppf':mf.magnitude_2_db*args,**kwargs)`

see `plot_uncertainty_bounds` for help

skrf.networkSet.NetworkSet.plot_uncertainty_decomposition

`NetworkSet.plot_uncertainty_decomposition(m=0, n=0)`

plots the total and component-wise uncertainty

Parameters `m` : int

first s-parameters index

`n` :

second s-parameter index

skrf.networkSet.NetworkSet.set_wise_function

`NetworkSet.set_wise_function(func, a_property, *args, **kwargs)`

calls a function on a specific property of the networks in this NetworkSet.

example: `my_ntwk_set.set_wise_func(mean,'s')`

skrf.networkSet.NetworkSet.signature

`NetworkSet.signature(m=0, n=0, from_mean=False, operation='__sub__', component='s_mag', vmax=None, *args, **kwargs)`

visualization of relative changes in a NetworkSet.

Creates a colored image representing the deviation of each Network from the from mean Network of the NetworkSet, vs frequency.

Parameters `m` : int

first s-parameters index

`n` : int

second s-parameter index

from_mean : Boolean

calculate distance from mean if True. or distance from first network in networkset if False.

operation : ['__sub__', '__div__'], ..

operation to apply between each network and the reference network, which is either the mean, or the initial ntwk.

component : ['s_mag','s_db','s_deg' ..]

scalar component of Network to plot on the imshow. should be a property of the Network object.

vmax : number

sets upper limit of colorbar, if None, will be set to 3*mean of the magnitude of the complex difference

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

passed to `imshow()`

skrf.networkSet.NetworkSet.uncertainty_ntwk_triplet

`NetworkSet.uncertainty_ntwk_triplet` (*attribute*, *n_deviations*=3)

returns a 3-tuple of Network objects which contain the mean, upper_bound, and lower_bound for the given Network attribute.

Used to save and plot uncertainty information data

3.2.3 frequency (**skrf.frequency**)

Provides a frequency object and related functions.

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

Frequency Class

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

skrf.frequency.Frequency

class `skrf.frequency.Frequency` (*start*, *stop*, *npoints*, *unit*='ghz', *sweep_type*='lin')

A frequency band.

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

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

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

Attributes

<code>center</code>	Center frequency.
<code>f</code>	Frequency vector in Hz
<code>f_scaled</code>	Frequency vector in units of <i>unit</i>
<code>multiplier</code>	Multiplier for forming axis
<code>multiplier_dict</code>	
<code>npoints</code>	starting frequency in Hz
<code>span</code>	the frequency span
<code>start</code>	starting frequency in Hz
<code>step</code>	the inter-frequency step size
<code>stop</code>	starting frequency in Hz
<code>unit</code>	Unit of this frequency band.
<code>unit_dict</code>	
<code>w</code>	Frequency vector in radians/s

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

Center frequency.

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

Frequency vector in Hz

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

The frequency vector in Hz

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

Multiplier for formatting axis

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

multiplier for this Frequencies unit

skrf.frequency.Frequency.multiplier_dict**Frequency.multiplier_dict** = {'hz': 1, 'khz': 1000.0, 'mhz': 1000000.0, 'thz': 1000000000000.0, 'ghz': 1000000000.0}**skrf.frequency.Frequency.npoints****Frequency.npoints**

starting frequency in Hz

skrf.frequency.Frequency.span**Frequency.span**

the frequency span

skrf.frequency.Frequency.start**Frequency.start**

starting frequency in Hz

skrf.frequency.Frequency.step**Frequency.step**

the inter-frequency step size

skrf.frequency.Frequency.stop**Frequency.stop**

starting frequency in Hz

skrf.frequency.Frequency.unit**Frequency.unit**

Unit of this frequency band.

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

Setting this attribute is not case sensitive.

Returns unit : string

lower-case string representing the frequency units

skrf.frequency.Frequency.unit_dict**Frequency.unit_dict** = {'hz': 'Hz', 'khz': 'KHz', 'mhz': 'MHz', 'thz': 'THz', 'ghz': 'GHz'}**skrf.frequency.Frequency.w****Frequency.w**

Frequency vector in radians/s

The frequency vector in rad/s

Returns w : `numpy.ndarray`

The frequency vector in rad/s

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

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

skrf.frequency.Frequency.__init__**Frequency.__init__** (*start, stop, npoints, 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,

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.labelXAxisFrequency.**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`, optionalAxes on which to label the plot, defaults what is returned by `matplotlib.gca()`**3.2.4 plotting (skrf.plotting)**

This module provides general plotting functions.

Charts

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

skrf.plotting.smith`skrf.plotting.smith` (*smithR=1, chart_type='z', 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

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='Imag', 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

3.2.5 touchstone (`skrf.touchstone`)

This module provides a class to represent touchstone files.

This module was written by Werner Hoch.

touchstone Class

<code>touchstone(filename)</code>	class to read touchstone s-parameter files
-----------------------------------	--

`skrf.touchstone.touchstone`

class `skrf.touchstone.touchstone(filename)`

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

Methods

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

`skrf.touchstone.touchstone.__init__`

`touchstone.__init__(filename)`

skrf.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.touchstone.touchstone.get_noise_data

`touchstone.get_noise_data ()`

TODO: NIY

skrf.touchstone.touchstone.get_noise_names

`touchstone.get_noise_names ()`

TODO: NIY

skrf.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.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.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.touchstone.touchstone.load_file

`touchstone.load_file (filename)`

Load the touchstone file into the interal data structures
contains touchstone class

3.2.6 convenience (skrf.convenience)

Holds pre-initialized objects's and functions that are general conveniences.

Plotting

<code>save_all_figs([dir, format])</code>	Save all open Figures to disk.
Continued on next page	

Table 3.17 – continued from previous page

<code>add_markers_to_lines(ax, marker_list, ...)</code>	adds markers to existing lines on a plot
<code>legend_off(ax)</code>	turn off the legend for a given axes.
<code>func_on_all_figs(func, *args, **kwargs)</code>	runs a function after making all open figures current.

skrf.convenience.save_all_figs

`skrf.convenience.save_all_figs` (*dir*='.', *format*=['eps', 'pdf', '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.convenience.add_markers_to_lines

`skrf.convenience.add_markers_to_lines` (*ax*=None, *marker_list*=['o', 'D', 's', '+', 'x'],
markevery=10)

adds markers to existing lines on a plot

this is convenient if you have already have a plot made, but then need to add markers afterwards, so that it can be interpreted in black and white. The *markevery* argument makes the markers less frequent than the data, which is generally what you want.

Parameters *ax* : matplotlib.Axes

axis which to add markers to, defaults to `gca()`

marker_list : list of marker characters

see `matplotlib.plot` help for possible marker characters

markevery : int

markevery number of points with a marker.

skrf.convenience.legend_off

`skrf.convenience.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.convenience.func_on_all_figs

`skrf.convenience.func_on_all_figs` (*func*, **args*, ***kwargs*)

runs a function after making all open figures current.

useful if you need to change the properties of many open figures at once, like turn off the grid.

Parameters **func** : function
function to call
***args, **kwargs** : passed to func

Examples

```
>>> rf.func_on_all_figs(grid, alpha=.3)
```

IO

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

skrf.convenience.hfss_touchstone_2_media

`skrf.convenience.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')
```

skrf.convenience.hfss_touchstone_2_gamma_z0

`skrf.convenience.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')
```

General

<code>now_string()</code>	returns a unique sortable string, representing the current time
<code>find_nearest(array, value)</code>	find nearest value in array.
<code>find_nearest_index(array, value)</code>	find nearest value in array.

skrf.convenience.now_string

`skrf.convenience.now_string()`
 returns a unique sortable string, representing the current time
 nice for generating date-time stamps to be used in file-names

skrf.convenience.find_nearest

`skrf.convenience.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.convenience.find_nearest_index

`skrf.convenience.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> :

Pre-initialized Objects

Frequency Objects

These are predefined `Frequency` objects that correspond to standard waveguide bands. This information is taken from the VDI Application Note 1002¹⁸. The naming convention is `f_wr#` where '#' is the band number.

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

Media Objects

These are predefined `Media` objects that represent Standardized transmission line media's. This information

Rectangular Waveguide Media's `RectangularWaveguide` Objects for standard bands.

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

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. but listing them here makes more sense.

Shorthand	Full Object Name
<code>F</code>	<code>Frequency</code>
<code>N</code>	<code>Network</code>
<code>NS</code>	<code>NetworkSet</code>
<code>M</code>	<code>Media</code>
<code>C</code>	<code>Calibration</code>

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

Shorthand	Full Object Name
<code>lat</code>	<code>load_all_touchstones()</code>
<code>saf</code>	<code>save_all_figs()</code>

¹⁸ VDI Application Note: VDI Waveguide Band Designations (VDI-1002) <http://vadiodes.com/VDI/pdf/waveguidechart200908.pdf>

References

3.2.7 mathFunctions (skrf.mathFunctions)

Provides commonly used mathematical functions.

Complex Component Conversion

<code>complex_2_reim(z)</code>	takes:
<code>complex_2_magnitude(input)</code>	returns the magnitude of a complex number.
<code>complex_2_db(input)</code>	returns the magnitude in dB of a complex number.
<code>complex_2_radian(input)</code>	returns the angle complex number in radians.
<code>complex_2_degree(input)</code>	returns the angle complex number in radians.
<code>complex_2_magnitude(input)</code>	returns the magnitude of a complex number.

skrf.mathFunctions.complex_2_reim

`skrf.mathFunctions.complex_2_reim(z)`

takes: input: complex number or array

return: real: real part of input imag: imaginary part of input

note: this just calls 'complex_components'

skrf.mathFunctions.complex_2_magnitude

`skrf.mathFunctions.complex_2_magnitude(input)`

returns the magnitude of a complex number.

skrf.mathFunctions.complex_2_db

`skrf.mathFunctions.complex_2_db(input)`

returns the magnitude in dB of a complex number.

returns: $20 \cdot \log_{10}(|z|)$

where z is a complex number

skrf.mathFunctions.complex_2_radian

`skrf.mathFunctions.complex_2_radian(input)`

returns the angle complex number in radians.

skrf.mathFunctions.complex_2_degree

`skrf.mathFunctions.complex_2_degree(input)`

returns the angle complex number in radians.

skrf.mathFunctions.complex_2_magnitude

`skrf.mathFunctions.complex_2_magnitude(input)`
returns the magnitude of a complex number.

Phase Unwrapping

<code>unwrap_rad(input)</code>	unwraps a phase given in radians
<code>sqrt_phase_unwrap(input)</code>	takes the square root of a complex number with unwrapped phase

skrf.mathFunctions.unwrap_rad

`skrf.mathFunctions.unwrap_rad(input)`
unwraps a phase given in radians

the normal numpy unwrap is not what you usually want for some reason

skrf.mathFunctions.sqrt_phase_unwrap

`skrf.mathFunctions.sqrt_phase_unwrap(input)`
takes the square root of a complex number with unwrapped phase

this idea came from Lihan Chen

Unit Conversion

<code>radian_2_degree(rad)</code>	
<code>degree_2_radian(deg)</code>	
<code>np_2_db(x)</code>	converts a value in dB to neper's
<code>db_2_np(x)</code>	converts a value in nepers to dB

skrf.mathFunctions.radian_2_degree

`skrf.mathFunctions.radian_2_degree(rad)`

skrf.mathFunctions.degree_2_radian

`skrf.mathFunctions.degree_2_radian(deg)`

skrf.mathFunctions.np_2_db

`skrf.mathFunctions.np_2_db(x)`
converts a value in dB to neper's

skrf.mathFunctions.db_2_np

`skrf.mathFunctions.db_2_np(x)`
 converts a value in nepers to dB

Scalar-Complex Conversion

These conversions are useful for wrapping other functions that dont support complex numbers.

<code>complex2Scalar(input)</code>
<code>scalar2Complex(input)</code>

skrf.mathFunctions.complex2Scalar

`skrf.mathFunctions.complex2Scalar(input)`

skrf.mathFunctions.scalar2Complex

`skrf.mathFunctions.scalar2Complex(input)`

Special Functions

<code>dirac_delta(x)</code>	the dirac function.
<code>neuman(x)</code>	neumans number
<code>null(A[, eps])</code>	calculates the null space of matrix A.

skrf.mathFunctions.dirac_delta

`skrf.mathFunctions.dirac_delta(x)`
 the dirac function.
 can take numpy arrays or numbers returns 1 or 0

skrf.mathFunctions.neuman

`skrf.mathFunctions.neuman(x)`
 neumans number
 2-dirac_delta(x)

skrf.mathFunctions.null

`skrf.mathFunctions.null(A, eps=1e-15)`
 calculates the null space of matrix A. i found this on stack overflow.

3.2.8 tlineFunctions (skrf.tlineFunctions)

This module provides functions related to transmission line theory.

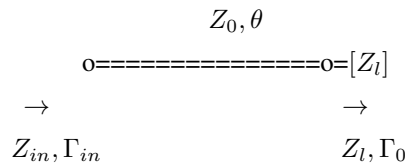
Impedance and Reflection Coefficient

These functions relate basic transmission line quantities such as characteristic impedance, input impedance, reflection coefficient, etc. Each function has two names. One is a long-winded but readable name and the other is a short-hand variable-like names. Below is a table relating these two names with each other as well as common mathematical symbols.

Symbol	Variable Name	Long Name
Z_l	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.



So, to clarify the confusion,

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

Short names

theta(gamma, f, d[, deg])	Calculates the electrical length of a section of transmission line.
z_l_2_Gamma0(z0, zl)	Returns the reflection coefficient for a given load impedance, and characteristic impedance.
Gamma0_2_zl(z0, Gamma)	calculates the input impedance given a reflection coefficient and characteristic impedance.
z_l_2_zin(z0, zl, theta)	input impedance of load impedance zl at a given electrical length, and characteristic impedance.
z_l_2_Gamma_in(z0, zl, theta)	reflection coefficient at a given electrical length, and characteristic impedance.
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 characteristic impedance.

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 characterisic impedance

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

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

complex reflection coefficient

z0 : number or array-like

characteristic impedance

Returns **zin** : number or array-like

input impedance

skrf.tlineFunctions.zl_2_zin

`skrf.tlineFunctions.zl_2_zin(z0, zl, theta)`

input impedance of load impedance zl at a given electrical length, given characteristic impedance z0.

Parameters **z0** : characteristic impedance.

zl : load impedance

theta : electrical length of the line, (may be complex)

skrf.tlineFunctions.zl_2_Gamma_in

`skrf.tlineFunctions.zl_2_Gamma_in(z0, zl, theta)`

skrf.tlineFunctions.Gamma0_2_Gamma_in

`skrf.tlineFunctions.Gamma0_2_Gamma_in(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 characterisic impedance of the medium Parameters ———

z0 - characteristic impedance. Gamma: reflection coefficient theta: electrical length of the line, (may be complex)

returns zin: input impedance at theta

Long-names

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

`skrf.tlineFunctions.distance_2_electrical_length`

`skrf.tlineFunctions.distance_2_electrical_length(gamma, f, d, deg=False)`

Calculates the electrical length of a section of transmission line.

$$\theta = \gamma(f) \cdot d$$

Parameters `gamma` : function

propagation constant function, which takes frequency in hz as a sole argument. see Notes.

`l` : number or array-like

length of line, in meters

`f` : number or array-like

frequency at which to calculate

`deg` : Boolean

return in degrees or not.

Returns `theta` : number or array-like

electrical length in radians or degrees, depending on value of deg.

See Also:

[`electrical_length_2_distance`](#) opposite conversion

Notes

the convention has been chosen that forward propagation is represented by the positive imaginary part of the value returned by the gamma function

`skrf.tlineFunctions.electrical_length_2_distance`

`skrf.tlineFunctions.electrical_length_2_distance(theta, gamma, f0, deg=True)`

Convert electrical length to a physical distance.

$$d = \frac{\theta}{\gamma(f_0)}$$

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

electrical 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` ($z0$, Γ)
calculates the input impedance given a reflection coefficient and characteristic impedance

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

Parameters Γ : number or array-like

complex reflection coefficient

$z0$: number or array-like

characteristic impedance

Returns z_{in} : number or array-like

input impedance

`skrf.tlineFunctions.reflection_coefficient_2_input_impedance_at_theta`

`skrf.tlineFunctions.reflection_coefficient_2_input_impedance_at_theta` ($z0$, Γ , θ)
calculates the input impedance at electrical length θ , given a reflection coefficient and characteristic impedance of the medium
Parameters ———

$z0$ - characteristic impedance. Γ : reflection coefficient θ : electrical length of the line, (may be complex)

returns z_{in} : input impedance at θ

`skrf.tlineFunctions.input_impedance_at_theta`

`skrf.tlineFunctions.input_impedance_at_theta` ($z0$, z_l , θ)
input impedance of load impedance z_l at a given electrical length, given characteristic impedance $z0$.

Parameters $z0$: characteristic impedance.

z_l : load impedance

θ : electrical length of the line, (may be complex)

`skrf.tlineFunctions.load_impedance_2_reflection_coefficient`

`skrf.tlineFunctions.load_impedance_2_reflection_coefficient` ($z0$, z_l)
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

z_l : 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)

Distributed Circuit and Wave Quantities

<code>distributed_circuit_2_propagation_impedance(...)</code>	Converts distributed circuit values to wave quantities.
<code>propagation_impedance_2_distributed_circuit(...)</code>	Converts wave quantities to distributed circuit values.

`skrf.tlineFunctions.distributed_circuit_2_propagation_impedance`

`skrf.tlineFunctions.distributed_circuit_2_propagation_impedance` (*distributed_admittance*,
dis-tributed_impedance)

Converts distributed circuit values to wave quantities.

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

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

Parameters `distributed_admittance` : number, array-like

distributed admittance

`distributed_impedance` : number, array-like

distributed impedance

Returns `propagation_constant` : number, array-like

distributed impedance

`characteristic_impedance` : number, array-like

distributed impedance

See Also:

`propagation_impedance_2_distributed_circuit` opposite conversion

skrf.tlineFunctions.propagation_impedance_2_distributed_circuit

`skrf.tlineFunctions.propagation_impedance_2_distributed_circuit` (*propagation_constant*,
characteristic_impedance)

Converts wave quantities to distributed circuit values.

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

$$Z' = \gamma Z_0 \quad Y' = \frac{\gamma}{Z_0}$$

Parameters **propagation_constant** : number, array-like

distributed impedance

characteristic_impedance : number, array-like

distributed impedance

Returns **distributed_admittance** : number, array-like

distributed admittance

distributed_impedance : number, array-like

distributed impedance

See Also:

`distributed_circuit_2_propagation_impedance` opposite conversion

Transmission Line Physics

<code>skin_depth(f, rho, mu_r)</code>	the skin depth for a material.
<code>surface_resistivity(f, rho, mu_r)</code>	surface resistivity.

skrf.tlineFunctions.skin_depth

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

see www.microwaves101.com for more info.

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

frequency, in Hz

rho : number or array-like

bulk resistivity of material, in ohm*m

mu_r : number or array-like

relative permeability of material

Returns **skin depth** : number or array-like

the skin depth, in m

skrf.tlineFunctions.surface_resistivity

skrf.tlineFunctions.**surface_resistivity** (*f*, *rho*, *mu_r*)
surface resistivity.

see www.microwaves101.com for more info.

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

frequency, in Hz

rho: number or array-like

bulk resistivity of material, in ohm*m

mu_r: number or array-like

relative permeability of material

Returns **surface resistivity: ohms/square** :

3.3 Packages

3.3.1 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.

Modules

calibration (skrf.calibration.calibration)

Contains the Calibration class, and supporting functions

`Calibration(measured, ideals[, type, ...])` An object to represent a VNA calibration instance.

Calibration Class

skrf.calibration.calibration.Calibration

class skrf.calibration.calibration.**Calibration** (*measured*, *ideals*, *type=None*,
is_reciprocal=False, *name=None*,
sloppy_input=False, ***kwargs*)

An object to represent a VNA calibration instance.

A Calibration object is used to perform a calibration given a set measurements and ideals responses. It can run a calibration, store results, and apply the results to calculate corrected measurements.

Attributes

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

skrf.calibration.calibration.Calibration.Ts`Calibration.Ts`

T-matrices used for de-embedding, a two-port calibration.

skrf.calibration.calibration.Calibration.caled_ntwk_sets`Calibration.caled_ntwk_sets`returns a NetworkSet for each `caled_ntwk`, based on their names**skrf.calibration.calibration.Calibration.caled_ntwks**`Calibration.caled_ntwks`

list of the calibrated, calibration standards.

skrf.calibration.calibration.Calibration.calibration_algorithm_dict`Calibration.calibration_algorithm_dict = {'two port': <function two_port at 0x3a38488>, '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 type 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 overdeterminded, this holds the residuals in the form of a vector.

also available are the complex residuals in the form of skrf.Network’s, see the property ‘residual_ntwks’

from numpy.linalg: residues: the sum of the residues; squared euclidean norm for each column vector in b (given ax=b)**skrf.calibration.calibration.Calibration.type****Calibration.type**

string representing what type of calibration is to be performed. supported types at the moment are:

‘one port’: standard one-port cal. if more than 2 measurement/ideal pairs are given it will calculate the least squares solution.**‘two port’:** two port calibration based on the error-box model

note: algorithms referenced by calibration_algorithm_dict, are stored in calibrationAlgorithms.py

Methods

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

Continued on next page

Table 3.31 – continued from previous page

<code>plot_residuals_smith</code>	see <code>plot_residuals</code>
<code>plot_uncertainty_per_standard</code>	Plots uncertainty associated with each calibration standard.
<code>run</code>	runs the calibration algorithm.
<code>total_error</code>	estimate of total error for overdetermined calibration with
<code>unbiased_error</code>	estimate of unbiased error for overdetermined calibration with
<code>uncertainty_per_standard</code>	given that you have repeat-connections of single standard,

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

Other Parameters `type` : string

the calibration algorithm. If *None*, the class will inspect number of ports on first *measured* `Network` and choose either ‘one port’ or ‘two port’. See **Notes** section for more info

`is_reciprocal` : Boolean

enables the reciprocity assumption on the calculation of the `error_network`, which is only relevant for one-port calibrations.

`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 ¹⁹ for explanation of what they are.

name: string :

the name of calibration, just for your convenience [None].

`sloppy_input` : Boolean.

Allows ideals and measured lists to be ‘aligned’ based on the network names

****kwargs** : key-word arguments

passed to the calibration algorithm, defined by *type*

Notes

All calibration algorithms are in stored in `skrf.calibration.calibrationAlgorithms`, refer to that file for documentation on the algorithms themselves. The `Calibration` class accesses those functions through the attribute ‘`calibration_algorithm_dict`’.

¹⁹ 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>

References

Examples

See the *Calibration* tutorial, or the examples sections for *One-Port Calibration* and *Two-Port Calibration*

`skrf.calibration.calibration.Calibration.apply_cal`

`Calibration.apply_cal(input_ntwk)`
apply the current calibration to a measurement.

takes:

input_ntwk: the measurement to apply the calibration to, a Network type.

returns: `caled`: the calibrated measurement, a Network type.

`skrf.calibration.calibration.Calibration.apply_cal_to_all_in_dir`

`Calibration.apply_cal_to_all_in_dir(dir, contains=None, f_unit='ghz')`
convenience function to apply calibration to an entire directory of measurements, and return a dictionary of the calibrated results, optionally the user can 'grep' the direction by using the contains switch.

takes: `dir`: directory of measurements (string) `contains`: will only load measurements who's filename contains this string.

f_unit: frequency unit, to use for all networks. see `frequency.Frequency.unit` for info.

returns:

ntwkDict: a dictionary of calibrated measurements, the keys are the filenames.

`skrf.calibration.calibration.Calibration.biased_error`

`Calibration.biased_error(std_names=None)`
estimate of biased error for overdetermined calibration with multiple connections of each standard

takes:

std_names: list of strings to uniquely identify each standard.*

returns:

systematic error: `skrf.Network` type who's `.s_mag` is proportional to the systematic error metric

note:

mathematically, this is $\text{mean}_s(|\text{mean}_c(\mathbf{r})|)$

where: `r`: complex residual errors `mean_c`: complex mean taken accross connection `mean_s`: complex mean taken accross standard

`skrf.calibration.calibration.Calibration.func_per_standard`

`Calibration.func_per_standard(func, attribute='s', std_names=None)`

`skrf.calibration.calibration.Calibration.mean_residuals`

`Calibration.mean_residuals()`

skrf.calibration.calibration.Calibration.plot_coefs_db

`Calibration.plot_coefs_db` (*ax=None, show_legend=True, **kwargs*)
 plot magnitude of the error coefficient dictionary

skrf.calibration.calibration.Calibration.plot_errors

`Calibration.plot_errors` (*std_names=None, scale='db', *args, **kwargs*)
 plot calibration error metrics for an over-determined calibration.
 see `biased_error`, `unbiased_error`, and `total_error` for more info

skrf.calibration.calibration.Calibration.plot_residuals

`Calibration.plot_residuals` (*attribute, *args, **kwargs*)
 plots a component of the residual errors on the Calibration-plane.

takes:

attribute: name of plotting method of Network class to call

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

***args, **kwargs:** passed to `plot_s_` 'attribute'()

note: the residuals are calculated by:

`(self.apply_cal(self.measured[k])-self.ideals[k])`

skrf.calibration.calibration.Calibration.plot_residuals_db

`Calibration.plot_residuals_db` (**args, **kwargs*)
 see `plot_residuals`

skrf.calibration.calibration.Calibration.plot_residuals_mag

`Calibration.plot_residuals_mag` (**args, **kwargs*)
 see `plot_residuals`

skrf.calibration.calibration.Calibration.plot_residuals_smith

`Calibration.plot_residuals_smith` (**args, **kwargs*)
 see `plot_residuals`

skrf.calibration.calibration.Calibration.plot_uncertainty_per_standard

`Calibration.plot_uncertainty_per_standard` (*scale='db', *args, **kwargs*)
 Plots uncertainty associated with each calibration standard.

This requires that each calibration standard is measured multiple times. The uncertainty associated with each standard is calculated by the complex standard deviation.

Parameters `scale` : 'db', 'lin'

plot uncertainties on linear or log scale

***args, **kwargs** : passed to `uncertainty_per_standard()`

See Also:

`uncertainty_per_standard()`

skrf.calibration.calibration.Calibration.run**Calibration.run()**

runs the calibration algorithm.

this is automatically called the first time any dependent property is referenced (like `error_ntwk`), but only the first time. if you change something and want to re-run the calibration

use this.

skrf.calibration.calibration.Calibration.total_error**Calibration.total_error** (*std_names=None*)

estimate of total error for overdetermined calibration with multiple connections of each standard. This is the combined effects of both biased and un-biased errors

takes:

std_names: list of strings to uniquely identify each standard.*

returns:

composit error: `skrf.Network` type who's `.s_mag` is proportional to the composit error metric

note:

mathematically, this is $\text{std_cs}(r)$

where: `r`: complex residual errors `std_cs`: standard deviation taken accross connections and standards

skrf.calibration.calibration.Calibration.unbiased_error**Calibration.unbiased_error** (*std_names=None*)

estimate of unbiased error for overdetermined calibration with multiple connections of each standard

takes:

std_names: list of strings to uniquely identify each standard.*

returns:

stochastic error: `skrf.Network` type who's `.s_mag` is proportional to the stochastic error metric

see also: `uncertainty_per_standard`, for this a measure of unbiased errors for each standard

note:

mathematically, this is $\text{mean_s}(\text{std_c}(r))$

where: `r`: complex residual errors `std_c`: standard deviation taken accross connections `mean_s`: complex mean taken accross standards

skrf.calibration.calibration.Calibration.uncertainty_per_standard**Calibration.uncertainty_per_standard** (*std_names=None, attribute='s'*)

given that you have repeat-connections of single standard, this calculates the complex standard deviation (distance) for each standard in the calibration across connection #.

takes:

std_names: list of strings to uniquely identify each standard.*

attribute: string passed to `func_on_networks` to calculate std deviation on a component if desired.
[`'s'`]

returns: list of `skrf.Networks`, whose magnitude of s-parameters is proportional to the standard deviation for that standard

***example:**

if your calibration had ideals named like: 'short 1', 'short 2', 'open 1', 'open 2', etc.

you would pass this `mycal.uncertainty_per_standard(['short','open','match'])`

calibrationAlgorithms (`skrf.calibration.calibrationAlgorithms`)

Contains calibrations algorithms and related functions, which are used in the `Calibration` class.

<code>one_port(measured, ideals)</code>	Standard algorithm for a one port calibration.
<code>one_port_nls(measured, ideals)</code>	one port non-linear least squares.
<code>two_port(measured, ideals[, switch_terms])</code>	Two port calibration based on the 8-term error model.
<code>parameterized_self_calibration(measured, ideals)</code>	An iterative, general self-calibration routine.
<code>parameterized_self_calibration_nls(measured, ...)</code>	An iterative, general self-calibration routine.

Calibration Algorithms

`skrf.calibration.calibrationAlgorithms.one_port`

`skrf.calibration.calibrationAlgorithms.one_port` (*measured, ideals*)

Standard algorithm for a one port calibration.

If more than three standards are supplied then a least square algorithm is applied.

Parameters **measured** : list of `Network` objects or `numpy.ndarray`

a list of the measured reflection coefficients. The elements of the list can either a `kxnxn` `numpy.ndarray`, representing a s-matrix, or list of 1-port `Network` objects.

ideals : list of `Network` objects or `numpy.ndarray`

a list of the ideal reflection coefficients. The elements of the list can either a `kxnxn` `numpy.ndarray`, representing a s-matrix, or list of 1-port `Network` objects.

Returns **output** : a dictionary

output information from the calibration, the keys are

- 'error coefficients': dictionary containing standard error coefficients
- 'residuals': a matrix of residuals from the least squared calculation. see `numpy.linalg.lstsq()` for more info

See Also:

`one_port_nls` for a non-linear least square implementation

Notes

uses `numpy.linalg.lstsq()` for least squares calculation

skrf.calibration.calibrationAlgorithms.one_port_nls

`skrf.calibration.calibrationAlgorithms.one_port_nls` (*measured*, *ideals*)
 one port non-linear least squares.

Parameters **measured** : list of `Network` objects or `numpy.ndarray`

a list of the measured reflection coefficients. The elements of the list can either a `kxnxn` `numpy.ndarray`, representing a s-matrix, or list of 1-port `Network` objects.

ideals : list of `Network` objects or `numpy.ndarray`

a list of the ideal reflection coefficients. The elements of the list can either a `kxnxn` `numpy.ndarray`, representing a s-matrix, or list of 1-port `Network` objects.

Returns **output** : a dictionary

a dictionary containing the following keys:

- ‘error coefficients’: dictionary containing standard error coefficients
- ‘residuals’: a matrix of residuals from the least squared calculation. see `numpy.linalg.lstsq()` for more info
- ‘cov_x’: covariance matrix

Notes

Uses `scipy.optimize.leastsq()` for non-linear least squares calculation

skrf.calibration.calibrationAlgorithms.two_port

`skrf.calibration.calibrationAlgorithms.two_port` (*measured*, *ideals*,
switch_terms=None)

Two port calibration based on the 8-term error model.

Takes two ordered lists of measured and ideal responses. Optionally, switch terms [R5] can be taken into account by passing a tuple containing the forward and reverse switch terms as 1-port `Network`s. This algorithm is based on the work in [R6] .

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 [R5] 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.virtualInstruments.vna`

References

[R5], [R6]

`skrf.calibration.calibrationAlgorithms.parameterized_self_calibration`

`skrf.calibration.calibrationAlgorithms.parameterized_self_calibration` (*measured*,
ide-
als,
show-
Progress=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*,
ide-
als_ps,
show-
Progress=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

<code>unterminate_switch_terms(two_port, gamma_f, ...)</code>	unterminates switch terms from raw measurements.
<code>abc_2_coefs_dict(abc)</code>	converts an abc ndarray to a dictionary containing the error
<code>eight_term_2_one_port_coefs(coefs)</code>	

Supporting Functions

skrf.calibration.calibrationAlgorithms.unterminate_switch_terms

`skrf.calibration.calibrationAlgorithms.unterminate_switch_terms` (*two_port*,
gamma_f,
gamma_r)

unterminates switch terms from raw measurements.

takes: *two_port*: the raw measurement, a 2-port `Network` type. *gamma_f*: the measured forward switch term, a

1-port Network type `gamma_r`: the measured reverse switch term, a 1-port Network type

returns: un-terminated measurement, a 2-port Network type

see: ‘Formulations of the Basic Vector Network Analyzer Error Model including Switch Terms’ by Roger B. Marks

`skrf.calibration.calibrationAlgorithms.abc_2_coefs_dict`

`skrf.calibration.calibrationAlgorithms.abc_2_coefs_dict(abc)`

converts an `abc` ndarray to a dictionary containing the error coefficients.

takes:

abc [Nx3 numpy.ndarray, which holds the complex calibration]

coefficients. the components of abc are $a[:] = abc[:,0]$ $b[:] = abc[:,1]$ $c[:] = abc[:,2]$,

a, b and c are related to the error network by $a = \det(e) = e_{01}e_{10} - e_{00}e_{11}$ $b = e_{00}$ $c = e_{11}$

returns:

coefsDict: dictionary containing the following ‘directivity’: e_{00} ‘reflection tracking’: $e_{01}e_{10}$ ‘source match’: e_{11}

note: e_{00} = directivity error $e_{10}e_{01}$ = reflection tracking error e_{11} = 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) [net-
work.plot_s_smith() for network in network_ensemble]
```

Classes

<code>Calibration(measured, ideals[, type, ...])</code>	An object to represent a VNA calibration instance.
---	--

skrf.calibration.calibration.Calibration

```
class skrf.calibration.calibration.Calibration (measured,          ideals,          type=None,
                                                is_reciprocal=False,      name=None,
                                                sloppy_input=False, **kwargs)
```

An object to represent a VNA calibration instance.

A Calibration object is used to perform a calibration given a set measurements and ideals responses. It can run a calibration, store results, and apply the results to calculate corrected measurements.

Attributes

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

skrf.calibration.calibration.Calibration.Ts

`Calibration.Ts`

T-matrices used for de-embedding, a two-port calibration.

skrf.calibration.calibration.Calibration.caled_ntwk_sets

`Calibration.caled_ntwk_sets`

returns a NetworkSet for each caled_ntwk, based on their names

skrf.calibration.calibration.Calibration.caled_ntwks

`Calibration.caled_ntwks`

list of the calibrated, calibration standards.

skrf.calibration.calibration.Calibration.calibration_algorithm_dict

`Calibration.calibration_algorithm_dict = {'two port': <function two_port at 0x3a38488>, '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 type 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

$$r = (E.\text{inv} ** m - i)$$

where, r: residual network, E: embedding network, m: measured network i: ideal network

This way the units of the residual networks are meaningful

note: the residuals are only calculated if they are not existent.

so, if you want to re-calculate the residual networks then you delete the property '_residual_ntwks'.

skrf.calibration.calibration.Calibration.residuals**Calibration.residuals**

if calibration is overdetermined, this holds the residuals in the form of a vector.

also available are the complex residuals in the form of skrf.Network's, see the property 'residual_ntwks'

from numpy.linalg: residues: the sum of the residues; squared euclidean norm for each column vector in b (given $ax=b$)

skrf.calibration.calibration.Calibration.type**Calibration.type**

string representing what type of calibration is to be performed. supported types at the moment are:

‘one port’: standard one-port cal. if more than 2 measurement/ideal pairs are given it will calculate the least squares solution.

‘two port’: two port calibration based on the error-box model

note: algorithms referenced by calibration_algorithm_dict, are stored in calibrationAlgorithms.py

Methods

<code>__init__</code>	Calibration initializer.
<code>apply_cal</code>	apply the current calibration to a measurement.
<code>apply_cal_to_all_in_dir</code>	convenience function to apply calibration to an entire directory
<code>biased_error</code>	estimate of biased error for overdetermined calibration with
<code>func_per_standard</code>	
<code>mean_residuals</code>	
<code>plot_coefs_db</code>	plot magnitude of the error coefficient dictionary
<code>plot_errors</code>	plot calibration error metrics for an over-determined calibration.
<code>plot_residuals</code>	plots a component of the residual errors on the Calibration-plane.
<code>plot_residuals_db</code>	see plot_residuals
<code>plot_residuals_mag</code>	see plot_residuals
<code>plot_residuals_smith</code>	see plot_residuals
<code>plot_uncertainty_per_standard</code>	Plots uncertainty associated with each calibration standard.
<code>run</code>	runs the calibration algorithm.
<code>total_error</code>	estimate of total error for overdetermined calibration with
<code>unbiased_error</code>	estimate of unbiased error for overdetermined calibration with
<code>uncertainty_per_standard</code>	given that you have repeat-connections of single standard,

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

Other Parameters **type** : string

the calibration algorithm. If *None*, the class will inspect number of ports on first *measured* `Network` and choose either *‘one port’* or *‘two port’*. See **Notes_** section for more infor

is_reciprocal : Boolean

enables the reciprocity assumption on the calculation of the *error_network*, which is only relevant for one-port calibrations.

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 ²⁰ for explanation of what they are.

name: string :

the name of calibration, just for your convenience [None].

sloppy_input : Boolean.

Allows ideals and measured lists to be 'aligned' based on the network names

****kwargs** : key-word arguments

passed to the calibration algorithm, defined by *type*

Notes

All calibration algorithms are in stored in `skrf.calibration.calibrationAlgorithms`, refer to that file for documentation on the algorithms themselves. The Calibration class accesses those functions through the attribute '`calibration_algorithm_dict`'.

References

Examples

See the *Calibration* tutorial, or the examples sections for *One-Port Calibration* and *Two-Port 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: `calcd`: the calibrated measurement, a `Network` type.

`skrf.calibration.calibration.Calibration.apply_cal_to_all_in_dir`

`Calibration.apply_cal_to_all_in_dir` (*dir*, *contains=None*, *f_unit='ghz'*)

convenience function to apply calibration to an entire directory of measurements, and return a dictionary of the calibrated results, optionally the user can 'grep' the direction by using the contains switch.

takes: *dir*: directory of measurements (string) *contains*: will only load measurements who's filename contains this string.

f_unit: frequency unit, to use for all networks. see `frequency.Frequency.unit` for info.

returns:

ntwkDict: a dictionary of calibrated measurements, the keys are the filenames.

²⁰ 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>

skrf.calibration.calibration.Calibration.biased_error**Calibration.biased_error** (*std_names=None*)

estimate of biased error for overdetermined calibration with multiple connections of each standard

takes:**std_names:** list of strings to uniquely identify each standard.***returns:****systematic error:** skrf.Network type who's **s_mag** is proportional to the systematic error metric**note:****mathematically, this is** $\text{mean}_s(|\text{mean}_c(r)|)$ **where:** **r:** complex residual errors **mean_c:** complex mean taken accross connection **mean_s:** complex mean taken accross standard**skrf.calibration.calibration.Calibration.func_per_standard****Calibration.func_per_standard** (*func, attribute='s', std_names=None*)**skrf.calibration.calibration.Calibration.mean_residuals****Calibration.mean_residuals** ()**skrf.calibration.calibration.Calibration.plot_coefs_db****Calibration.plot_coefs_db** (*ax=None, show_legend=True, **kwargs*)

plot magnitude of the error coefficient dictionary

skrf.calibration.calibration.Calibration.plot_errors**Calibration.plot_errors** (*std_names=None, scale='db', *args, **kwargs*)

plot calibration error metrics for an over-determined calibration.

see **biased_error**, **unbiased_error**, and **total_error** for more info**skrf.calibration.calibration.Calibration.plot_residuals****Calibration.plot_residuals** (*attribute, *args, **kwargs*)

plots a component of the residual errors on the Calibration-plane.

takes:**attribute:** name of plotting method of Network class to call**possible options are:** 'mag', 'db', 'smith', 'deg', etc***args,**kwargs:** passed to **plot_s_**'attribute'()**note:** the residuals are calculated by: $(\text{self.apply_cal}(\text{self.measured}[k]) - \text{self.ideals}[k])$ **skrf.calibration.calibration.Calibration.plot_residuals_db****Calibration.plot_residuals_db** (**args, **kwargs*)see **plot_residuals**

skrf.calibration.calibration.Calibration.plot_residuals_mag`Calibration.plot_residuals_mag(*args, **kwargs)`see `plot_residuals`**skrf.calibration.calibration.Calibration.plot_residuals_smith**`Calibration.plot_residuals_smith(*args, **kwargs)`see `plot_residuals`**skrf.calibration.calibration.Calibration.plot_uncertainty_per_standard**`Calibration.plot_uncertainty_per_standard(scale='db', *args, **kwargs)`

Plots uncertainty associated with each calibration standard.

This requires that each calibration standard is measured multiple times. The uncertainty associated with each standard is calculated by the complex standard deviation.

Parameters `scale` : 'db', 'lin'

plot uncertainties on linear or log scale

`*args, **kwargs` : passed to `uncertainty_per_standard()`

See Also:

`uncertainty_per_standard()`

skrf.calibration.calibration.Calibration.run`Calibration.run()`

runs the calibration algorithm.

this is automatically called the first time any dependent property is referenced (like `error_ntwk`), but only the first time. if you change something and want to re-run the calibration

use this.

skrf.calibration.calibration.Calibration.total_error`Calibration.total_error(std_names=None)`

estimate of total error for overdetermined calibration with multiple connections of each standard. This is the combined effects of both biased and un-biased errors

takes:

std_names: list of strings to uniquely identify each standard.*

returns:

composit error: `skrf.Network` type who's `.s_mag` is proportional to the composit error metric

note:

mathematically, this is `std_cs(r)`

where: `r`: complex residual errors `std_cs`: standard deviation taken accross connections and standards

skrf.calibration.calibration.Calibration.unbiased_error

`Calibration.unbiased_error` (*std_names=None*)

estimate of unbiased error for overdetermined calibration with multiple connections of each standard

takes:

std_names: list of strings to uniquely identify each standard.*

returns:

stochastic error: `skrf.Network` type who's `.s_mag` is proportional to the stochastic error metric

see also: `uncertainty_per_standard`, for this a measure of unbiased errors for each standard

note:

mathematically, this is $\text{mean}_s(\text{std}_c(r))$

where: `r`: complex residual errors `std_c`: standard deviation taken accross connections `mean_s`: complex mean taken accross standards

skrf.calibration.calibration.Calibration.uncertainty_per_standard

`Calibration.uncertainty_per_standard` (*std_names=None, attribute='s'*)

given that you have repeat-connections of single standard, this calculates the complex standard deviation (distance) for each standard in the calibration across connection #.

takes:

std_names: list of strings to uniquely identify each standard.*

attribute: string passed to `func_on_networks` to calculate std deviation on a component if desired. `['s']`

returns: list of `skrf.Networks`, whose magnitude of s-parameters is proportional to the standard deviation for that standard

***example:**

if your calibration had ideals named like: 'short 1', 'short 2', 'open 1', 'open 2', etc.

you would pass this `mycal.uncertainty_per_standard(['short','open','match'])`

3.3.2 media (skrf.media)

This package provides objects representing transmission line mediums.

The `Media` object is the base-class that is inherited by specific transmission line instances, such as `Freespace`, or `RectangularWaveguide`. The `Media` object provides generic methods to produce `Network`'s for any transmission line medium, such as `line()` and `delay_short()`. These methods are inherited by the specific transmission line classes, which 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

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 transmission line medium, such as `line()` and `delay_short()`.

The initializer for this class has flexible argument types. This allows for the important attributes of the **Media** object to be dynamic. For example, if a **Media** object's propagation constant is a function of some attribute of that object, say *conductor_width*, then the propagation constant will change when that attribute changes. See `__init__()` for details.

The network creation methods build off of each other. For example, the special load cases, such as `short()` and `open()` call `load()` with given arguments for Γ_0 , and the **delay_** and **shunt_** functions call `line()` and `shunt()` respectively. This minimizes re-implementation.

Most methods initialize the **Network** by calling `match()` to create a 'blank' **Network**, and then fill in the s-matrix.

Attributes

<code>characteristic_impedance</code>	Characterisitic impedance
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

skrf.media.media.Media.characteristic_impedance

Media.characteristic_impedance

Characterisitic impedance

The *characteristic_impedance* can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns *characteristic_impedance* : `numpy.ndarray`

skrf.media.media.Media.propagation_constant

Media.propagation_constant

Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns *propagation_constant* : `numpy.ndarray`

complex propagation constant for this media

Notes

propagation_constant must adhere to the following convention,

- $\text{positive real}(\text{propagation_constant}) = \text{attenuation}$
- $\text{positive imag}(\text{propagation_constant}) = \text{forward propagation}$

skrf.media.media.Media.z0**Media.z0**

Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is *None* then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns `port_impedance` : `numpy.ndarray`

the media's port impedance

Methods

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

skrf.media.media.Media.__init__**Media.__init__** (*frequency*, *propagation_constant*, *characteristic_impedance*, *z0=None*)

The Media initializer.

This initializer has flexible argument types. The parameters *propagation_constant*, *characteristic_impedance*

and z_0 can all be either static or dynamic. This is achieved by allowing those arguments to be either:

- functions which take no arguments or
- values (numbers or arrays)

In the case where the media's propagation constant may change after initialization, because you adjusted a parameter of the media, then passing the `propagation_constant` as a function allows it to change when the media's parameters do.

Parameters `frequency` : `Frequency` object

frequency band of this transmission line medium

propagation_constant : number, array-like, or a function

propagation constant for the medium.

characteristic_impedance : number, array-like, or a function

characteristic impedance of transmission line medium.

z0 : number, array-like, or a function

the port impedance for media, IF its different from the characteristic impedance of the transmission line medium (None) [a number]. if $z_0 = \text{None}$ then will set to characteristic_impedance

See Also:

`from_csv()` function to create a Media object from a csv file containing gamma/ z_0

Notes

propagation_constant must adhere to the following convention,

- positive $\text{real}(\text{gamma})$ = attenuation
- positive $\text{imag}(\text{gamma})$ = forward propagation

the z_0 parameter is needed in some cases. For example, the `RectangularWaveguide` is an example where you may need this, because the characteristic impedance is frequency dependent, but the touchstone's created by most VNA's have $z_0=1$

skrf.media.media.Media.capacitor

`Media.capacitor(C, **kwargs)`

Capacitor

Parameters `C` : number, array

Capacitance, in Farads. If this is an array, must be of same length as frequency vector.

****kwargs** : key word 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 argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **delay_load** : `Network` object

a delayed load

See Also:`line` creates the network for line`load` creates the network for the load**Notes**

This calls

```
line(d,unit, **kwargs) ** load(Gamma0, **kwargs)
```

Examples

```
>>> my_media.delay_load(-.5, 90, 'deg', z0=50)
```

skrf.media.media.Media.delay_open`Media.delay_open (d, unit='m', **kwargs)`

Delayed open transmission line

Parameters **d** : number

the length of transmissin line (see unit argument)

unit : ['m','deg','rad']**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)

- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `delay_open` : `Network` object

a delayed open

See Also:

`delay_load` `delay_short` just calls this function

skrf.media.media.Media.delay_short

`Media.delay_short` (*d*, *unit*='m', ****kwargs**)

Delayed Short

A transmission line of given length terminated with a short.

Parameters *d* : number

the length of transmissin line (see unit argument)

unit : ['m','deg','rad']

the units of d. possible options are:

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `delay_short` : `Network` object

a delayed short

See Also:

`delay_load` `delay_short` just calls this function

skrf.media.media.Media.electrical_length

`Media.electrical_length` (*d*, *deg*=*False*)

calculates the electrical length for a given distance, at the center frequency.

Parameters *d*: number or array-like :

delay distance, in meters

deg: Boolean :

return 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 argumentspassed to `match()`, which is called initially to create a ‘blank’ network.**Returns** `missmatch` : `Network` object

a 2-port network representing the impedance mismatch

See Also:`match` called to create a ‘blank’ network**Notes**If `z1` and `z2` are arrays, they must be of same length as the `Media.frequency.npoints`**skrf.media.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 argumentspassed 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: kxn, where k is #frequency points in media, and n is *nports*

nports : int

number of ports

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns *load* :class:‘~skrf.network.Network’ object :

n-port load, where $S = \text{Gamma0} * \text{eye}(\dots)$

skrf.media.media.Media.match`Media.match` (*nports*=1, *z0*=None, ***kwargs*)Perfect matched load ($\Gamma_0 = 0$).**Parameters** *nports* : int

number of ports

z0 : number, or array-likecharacteristic impedance. Default is None, in which case the Media's *z0* is used. This sets the resultant Network's *z0*.***kwargs* : key word argumentspassed 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 argumentspassed 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 argumentspassed 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 argumentspassed 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 argumentspassed to `tee()`**Returns** `shunted_ntwk` : `Network` objecta shunted a ntwk. The resultant shunted_ntwk will have $(2 + \text{ntwk.number_of_ports} - 1)$ ports.**skrf.media.media.Media.shunt_capacitor**`Media.shunt_capacitor (C, *args, **kwargs)`

Shunted capacitor

Parameters `C` : number, array-like

Capacitance in Farads.

***args,**kwargs** : arguments, keyword argumentspassed to func:`delay_open`**Returns** `shunt_capacitor` : `Network` object

shunted capacitor(2-port)

Notes

This calls:

`shunt(capacitor(C, *args, **kwargs))`

skrf.media.media.Media.shunt_delay_load

`Media.shunt_delay_load(*args, **kwargs)`

Shunted delayed load

Parameters `*args, **kwargs` : arguments, keyword arguments
passed to func:*delay_load*

Returns `shunt_delay_load` : `Network` object
a shunted delayed load (2-port)

Notes

This calls:

```
shunt(delay_load(*args, **kwargs))
```

skrf.media.media.Media.shunt_delay_open

`Media.shunt_delay_open(*args, **kwargs)`

Shunted delayed open

Parameters `*args, **kwargs` : arguments, keyword arguments
passed to func:*delay_open*

Returns `shunt_delay_open` : `Network` object
shunted delayed open (2-port)

Notes

This calls:

```
shunt(delay_open(*args, **kwargs))
```

skrf.media.media.Media.shunt_delay_short

`Media.shunt_delay_short(*args, **kwargs)`

Shunted delayed short

Parameters `*args, **kwargs` : arguments, keyword arguments
passed to func:*delay_open*

Returns `shunt_delay_load` : `Network` object
shunted delayed open (2-port)

Notes

This calls:

```
shunt(delay_short(*args, **kwargs))
```

skrf.media.media.Media.shunt_inductor`Media.shunt_inductor(L, *args, **kwargs)`

Shunted inductor

Parameters `L` : number, array-like

Inductance in Farads.

***args, **kwargs** : arguments, keyword argumentspassed 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 argumentspassed 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 argumentspassed to `match()`, which is called initially to create a ‘blank’ network.**Returns** `tee` : `Network` object

a 3-port splitter

See Also:`splitter` this just calls `splitter(3)``match` called to create a ‘blank’ network

skrf.media.media.Media.theta_2_d`Media.theta_2_d(theta, deg=True)`

Converts electrical length to physical distance.

The given electrical length is to be at the center frequency.

Parameters **theta** : number

electrical length, at band center (see `deg` for unit)

deg : Boolean

is theta in degrees?

Returns **d** : number

physical distance in meters

skrf.media.media.Media.thru`Media.thru(**kwargs)`

Matched transmission line of length 0.

Parameters ****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns **thru** : `Network` object

matched transmission line of 0 length

See Also:

`line` this just calls `line(0)`

skrf.media.media.Media.white_gaussian_polar`Media.white_gaussian_polar(phase_dev, mag_dev, n_ports=1, **kwargs)`

Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

Parameters **phase_mag** : number

standard deviation of magnitude

phase_dev : number

standard deviation of phase

n_ports : int

number of ports.

****kwargs** : passed to `Network`

initializer

Returns **result** : `Network` object

a noise network

skrf.media.media.Media.write_csv

`Media.write_csv` (*filename*='f, gamma, z0.csv')

write this media's frequency, z0, and gamma to a csv file.

Parameters *filename* : string

file name to write out data to

See Also:

`from_csv` class method to initialize Media object from a csv file written from this function

Transmission Line Classes

<code>DistributedCircuit</code>	Generic, distributed circuit TEM transmission line
<code>RectangularWaveguide</code>	Rectangular Waveguide medium.
<code>CPW</code>	Coplanar waveguide class
<code>Freespace</code>	Represents a plane-wave in a homogeneous freespace, defined by the space's relative permativity and

skrf.media.distributedCircuit.DistributedCircuit

class `skrf.media.distributedCircuit.DistributedCircuit` (*frequency*, *C*, *I*, *R*, *G*, *args, **kwargs)

Generic, distributed circuit TEM transmission line

A TEM transmission line, defined in terms of distributed impedance and admittance values. A Distributed Circuit may be defined in terms of the following attributes,

Quantity	Symbol	Property
Distributed Capacitance	C'	C
Distributed Inductance	I'	I
Distributed Resistance	R'	R
Distributed Conductance	G'	G

From these, the following quantities may be calculated, which are functions of angular frequency (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	$Z_0 = \sqrt{\frac{Z'}{Y'}}$	<code>z0()</code>
Propagation Constant	$\gamma = \sqrt{Z'Y'}$	<code>gamma()</code>

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

$$+\Re\{\gamma\} = \text{attenuation}$$

$$-\Im\{\gamma\} = \text{forward propagation}$$

Attributes

<code>Y</code>	Distributed Admittance, Y'
<code>Z</code>	Distributed Impedance, Z'
<code>characteristic_impedance</code>	Characterisitic impedance
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

skrf.media.distributedCircuit.DistributedCircuit.YDistributedCircuit.**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.ZDistributedCircuit.**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_impedanceDistributedCircuit.**characteristic_impedance**

Characterisitic impedance

The characteristic_impedance can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns **characteristic_impedance** : numpy.ndarray**skrf.media.distributedCircuit.DistributedCircuit.propagation_constant**DistributedCircuit.**propagation_constant**

Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns **propagation_constant** : numpy.ndarray

complex propagation constant for this media

Notes

propagation_constant must adhere to the following convention,

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

skrf.media.distributedCircuit.DistributedCircuit.z0

DistributedCircuit.z0

Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is `None` then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns `port_impedance` : `numpy.ndarray`
the media's port impedance

Methods

<code>z0</code>	Characteristic Impedance, Z_0
<code>__init__</code>	Distributed Circuit constructor.
<code>capacitor</code>	Capacitor
<code>delay_load</code>	Delayed load
<code>delay_open</code>	Delayed open transmission line
<code>delay_short</code>	Delayed Short
<code>electrical_length</code>	calculates the electrical length for a given distance, at
<code>from_Media</code>	Initializes a DistributedCircuit from an existing
<code>from_csv</code>	create a Media from numerical values stored in a csv file.
<code>gamma</code>	Propagation Constant, γ
<code>guess_length_of_delay_short</code>	Guess physical length of a delay short.
<code>impedance_mismatch</code>	Two-port network for an impedance miss-match
<code>inductor</code>	Inductor
<code>line</code>	Matched transmission line of given length
<code>load</code>	Load of given reflection coefficient.
<code>match</code>	Perfect matched load ($\Gamma_0 = 0$).
<code>open</code>	Open ($\Gamma_0 = 1$)
<code>resistor</code>	Resistor
<code>short</code>	Short ($\Gamma_0 = -1$)
<code>shunt</code>	Shunts a <code>Network</code>
<code>shunt_capacitor</code>	Shunted capacitor
<code>shunt_delay_load</code>	Shunted delayed load
<code>shunt_delay_open</code>	Shunted delayed open
<code>shunt_delay_short</code>	Shunted delayed short
<code>shunt_inductor</code>	Shunted inductor

Continued on next page

Table 3.43 – continued from previous page

<code>splitter</code>	Ideal, lossless n-way splitter.
<code>tee</code>	Ideal, lossless tee.
<code>theta_2_d</code>	Converts electrical length to physical distance.
<code>thru</code>	Matched transmission line of length 0.
<code>white_gaussian_polar</code>	Complex zero-mean gaussian white-noise network.
<code>write_csv</code>	write this media’s frequency, <code>z0</code> , and <code>gamma</code> to a csv file.

skrf.media.distributedCircuit.DistributedCircuit.Z0`DistributedCircuit.Z0()`Characteristic Impedance, Z_0

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

Returns `Z0` : `numpy.ndarray`

Characteristic Impedance in units of ohms

skrf.media.distributedCircuit.DistributedCircuit.__init__`DistributedCircuit.__init__(frequency, C, I, R, G, *args, **kwargs)`

Distributed Circuit constructor.

Parameters `frequency` : `Frequency` object`C` : number, or array-like

distributed capacitance, in F/m

`I` : number, or array-like

distributed inductance, in H/m

`R` : number, or array-like

distributed resistance, in Ohm/m

`G` : number, or array-like

distributed conductance, in S/m

Notes`C,I,R,G` can all be vectors as long as they are the same lengthThis 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 argumentspassed to `match()`, which is called initially to create a ‘blank’ network.

Returns `capacitor` : a 2-port `Network`

See Also:

`match` function called to create a ‘blank’ network

`skrf.media.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 argumentspassed 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 argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **delay_short** : `Network` object

a delayed short

See Also:`delay_load` `delay_short` just calls this function

skrf.media.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 electal 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 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 argumentspassed to `match()`, which is called initially to create a ‘blank’ network.**Returns** `mismatch` : `Network` object

a 2-port network representing the impedance mismatch

See Also:`match` called to create a ‘blank’ network**Notes**If `z1` and `z2` are arrays, they must be of same length as the `Media.frequency.npoints`**skrf.media.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 argumentspassed 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 transmission 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: kxn, where k is #frequency points in media, and n is *nports*

nports : int

number of ports

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `load` :class:‘~skrf.network.Network’ object :

n-port load, where $S = \text{Gamma0} * \text{eye}(\dots)$

skrf.media.distributedCircuit.DistributedCircuit.match

`DistributedCircuit.match` (*nports*=1, *z0*=None, ****kwargs**)

Perfect matched load ($\Gamma_0 = 0$).

Parameters **nports** : int

number of ports

z0 : number, or array-like

characteristic impedance. Default is None, in which case the Media’s *z0* is used. This sets the resultant Network’s *z0*.

****kwargs** : key word arguments

passed to `Network` initializer

Returns `match` : `Network` object

a n-port match

Examples

```
>>> my_match = my_media.match(2, z0 = 50, name='Super Awesome Match')
```

skrf.media.distributedCircuit.DistributedCircuit.open`DistributedCircuit.open` (*nports=1*, ***kwargs*)Open ($\Gamma_0 = 1$)**Parameters** *nports* : int

number of ports

****kwargs** : key word argumentspassed 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 argumentspassed 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 argumentspassed 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.shuntDistributedCircuit.**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 argumentspassed to `tee()`**Returns** *shunted_ntwk* : *Network* objecta shunted a *ntwk*. The resultant *shunted_ntwk* will have $(2 + \text{ntwk.number_of_ports} - 1)$ ports.**skrf.media.distributedCircuit.DistributedCircuit.shunt_capacitor**DistributedCircuit.**shunt_capacitor** (*C*, ***args**, ****kwargs**)

Shunted capacitor

Parameters *C* : number, array-like

Capacitance in Farads.

args, **kwargs** : arguments, keyword argumentspassed to func:*delay_openReturns** *shunt_capacitor* : *Network* object

shunted capacitor(2-port)

Notes

This calls:

`shunt(capacitor(C, *args, **kwargs))`**skrf.media.distributedCircuit.DistributedCircuit.shunt_delay_load**DistributedCircuit.**shunt_delay_load** (***args**, ****kwargs**)

Shunted delayed load

Parameters ***args, **kwargs** : arguments, keyword argumentspassed to func:*delay_load***Returns** *shunt_delay_load* : *Network* object

a shunted delayed load (2-port)

Notes

This calls:

`shunt(delay_load(*args, **kwargs))`

skrf.media.distributedCircuit.DistributedCircuit.shunt_delay_open**DistributedCircuit.shunt_delay_open** (*args, **kwargs)

Shunted delayed open

Parameters *args,**kwargs : arguments, keyword arguments
passed to func:*delay_open*

Returns **shunt_delay_open** : *Network* object
shunted delayed open (2-port)

Notes

This calls:

```
shunt(delay_open(*args, **kwargs))
```

skrf.media.distributedCircuit.DistributedCircuit.shunt_delay_short**DistributedCircuit.shunt_delay_short** (*args, **kwargs)

Shunted delayed short

Parameters *args,**kwargs : arguments, keyword arguments
passed to func:*delay_open*

Returns **shunt_delay_load** : *Network* object
shunted delayed open (2-port)

Notes

This calls:

```
shunt(delay_short(*args, **kwargs))
```

skrf.media.distributedCircuit.DistributedCircuit.shunt_inductor**DistributedCircuit.shunt_inductor** (L, *args, **kwargs)

Shunted inductor

Parameters **L** : number, array-like
Inductance in Farads.

*args,**kwargs : arguments, keyword arguments
passed to func:*delay_open*

Returns **shunt_inductor** : *Network* object
shunted inductor(2-port)

Notes

This calls:


```
shunt(inductor(C,*args, **kwargs))
```

skrf.media.distributedCircuit.DistributedCircuit.splitter

DistributedCircuit.**splitter**(*nports*, ***kwargs*)

Ideal, lossless n-way splitter.

Parameters *nports* : int

number of ports

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns *tee* : `Network` object

a n-port splitter

See Also:

`match` called to create a ‘blank’ network

skrf.media.distributedCircuit.DistributedCircuit.tee

DistributedCircuit.**tee**(***kwargs*)

Ideal, lossless tee. (3-port splitter)

Parameters ***kwargs* : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns *tee* : `Network` object

a 3-port splitter

See Also:

`splitter` this just calls `splitter(3)`

`match` called to create a ‘blank’ network

skrf.media.distributedCircuit.DistributedCircuit.theta_2_d

DistributedCircuit.**theta_2_d**(*theta*, *deg=True*)

Converts electrical length to physical distance.

The given electrical length is to be at the center frequency.

Parameters *theta* : number

electrical length, at band center (see *deg* for unit)

deg : Boolean

is theta in degrees?

Returns *d* : number

physical distance in meters

skrf.media.distributedCircuit.DistributedCircuit.thru`DistributedCircuit.thru(**kwargs)`

Matched transmission line of length 0.

Parameters `**kwargs` : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `thru` : `Network` object

matched transmission line of 0 length

See Also:

`line` this just calls `line(0)`

skrf.media.distributedCircuit.DistributedCircuit.white_gaussian_polar`DistributedCircuit.white_gaussian_polar(phase_dev, mag_dev, n_ports=1, **kwargs)`

Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

Parameters `phase_mag` : number

standard deviation of magnitude

`phase_dev` : number

standard deviation of phase

`n_ports` : int

number of ports.

`**kwargs` : passed to `Network`

initializer

Returns `result` : `Network` object

a noise network

skrf.media.distributedCircuit.DistributedCircuit.write_csv`DistributedCircuit.write_csv(filename='f, gamma, z0.csv')`

write this media’s frequency, `z0`, and `gamma` to a csv file.

Parameters `filename` : string

file name to write out data to

See Also:

`from_csv` class method to initialize Media object from a csv file written from this function

skrf.media.rectangularWaveguide.RectangularWaveguide

```
class skrf.media.rectangularWaveguide.RectangularWaveguide(frequency, a, b=None,
                                                             mode_type='te', m=1,
                                                             n=0, ep_r=1, mu_r=1,
                                                             *args, **kwargs)
```

Rectangular Waveguide medium.

Represents a single mode of a homogeneously filled rectangular waveguide of cross-section $a \times b$. The mode is determined by mode-type (te or tm) and mode indices (m and n).

Quantity	Symbol	Variable
Characteristic Wave Number	k_0	<code>k0</code>
Cut-off Wave Number	k_c	<code>kc</code>
Longitudinal Wave Number	k_z	<code>kz</code>
Transverse Wave Number (a)	k_x	<code>kx</code>
Transverse Wave Number (b)	k_y	<code>ky</code>
Characteristic Impedance	Z_0	<code>z0</code>

Attributes

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

`skrf.media.rectangularWaveguide.RectangularWaveguide.characteristic_impedance`

`RectangularWaveguide.characteristic_impedance`

Characterisitic impedance

The `characteristic_impedance` can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the characteristic 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

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

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 it must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns **propagation_constant** : `numpy.ndarray`
complex propagation constant for this media

Notes

propagation_constant must adhere to the following convention,

- `positive real(propagation_constant)` = attenuation
- `positive imag(propagation_constant)` = forward propagation

skrf.media.rectangularWaveguide.RectangularWaveguide.z0

RectangularWaveguide.z0

Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is `None` then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function it must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns `port_impedance` : `numpy.ndarray`

the media's port impedance

Methods

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

skrf.media.rectangularWaveguide.RectangularWaveguide.Z0

RectangularWaveguide.Z0()

The characteristic impedance

skrf.media.rectangularWaveguide.RectangularWaveguide.__init__
`RectangularWaveguide.__init__(frequency, a, b=None, mode_type='te', m=1, n=0, ep_r=1, mu_r=1, *args, **kwargs)`

RectangularWaveguide initializer

Parameters **frequency** : class:~skrf.frequency.Frequency object

frequency band for this media

a : number

width of waveguide, in meters.

b : number

height of waveguide, in meters. If *None* defaults to $a/2$

mode_type : ['te','tm']

mode type, transverse electric (te) or transverse magnetic (tm) to-z. where z is direction of propagation

m : int

mode index in 'a'-direction

n : int

mode index in 'b'-direction

ep_r : number, array-like,

filling material's relative permativity

mu_r : number, array-like

filling material's relative permeability

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

passed to `Media`'s constructor (`__init__()`)

Examples

Most common usage is standard aspect ratio (2:1) dominant mode, TE10 mode of wr10 waveguide can be constructed by

```
>>> freq = rf.Frequency(75,110,101,'ghz')
>>> rf.RectangularWaveguide(freq, 100*mil)
```

skrf.media.rectangularWaveguide.RectangularWaveguide.capacitor

`RectangularWaveguide.capacitor(C, **kwargs)`

Capacitor

Parameters **C** : number, array

Capacitance, in Farads. If this is an array, must be of same length as frequency vector.

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a 'blank' network.

Returns `capacitor` : a 2-port `Network`

See Also:

`match` function called to create a ‘blank’ network

`skrf.media.rectangularWaveguide.RectangularWaveguide.delay_load`

`RectangularWaveguide.delay_load` (*Gamma0*, *d*, *unit*=‘m’, ***kwargs*)

Delayed load

A load with reflection coefficient *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_openRectangularWaveguide.**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 argumentspassed 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 argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **delay_short** : `Network` object

a delayed short

See Also:`delay_load` `delay_short` just calls this function

skrf.media.rectangularWaveguide.RectangularWaveguide.electrical_lengthRectangularWaveguide.**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 electal 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_shortRectangularWaveguide.**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_mismatchRectangularWaveguide.**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 argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** *missmatch* : Network object

a 2-port network representing the impedance mismatch

See Also:`match` called to create a 'blank' network

Notes

If `z1` and `z2` are arrays, they must be of same length as the `Media.frequency.npoints`

`skrf.media.rectangularWaveguide.RectangularWaveguide.inductor`

`RectangularWaveguide.inductor` (*L*, ***kwargs*)

Inductor

Parameters *L* : number, array

Inductance, in Henrys. If this is an array, must be of same length as frequency vector.

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns *inductor* : a 2-port `Network`

See Also:

`match` function called to create a ‘blank’ network

`skrf.media.rectangularWaveguide.RectangularWaveguide.kz`

`RectangularWaveguide.kz` ()

The Longitudinal wave number, aka propagation constant.

Defined as

$$k_z = \pm \sqrt{k_0^2 - k_c^2}$$

This is.

- IMAGINARY for propagating modes
- REAL for non-propagating modes,

Returns *kz* : number

The propagation constant

`skrf.media.rectangularWaveguide.RectangularWaveguide.line`

`RectangularWaveguide.line` (*d*, *unit='m'*, ***kwargs*)

Matched transmission line of given length

The units of *length* are interpreted according to the value of *unit*.

Parameters *d* : number

the length of transmissin line (see unit argument)

unit : ['m','deg','rad']

the units of d. possible options are:

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `line` : `Network` object

matched transmission line of given length

Examples

```
>>> my_media.line(90, 'deg', z0=50)
```

`skrf.media.rectangularWaveguide.RectangularWaveguide.load`

`RectangularWaveguide.load` (*Gamma0*, *nports*=1, ***kwargs*)

Load of given reflection coefficient.

Parameters `Gamma0` : number, array-like

Reflection coefficient of load (linear, not in db). If its an array it must be of shape: $k \times n$, where k is #frequency points in media, and n is *nports*

`nports` : int

number of ports

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `load` :class:‘~skrf.network.Network’ object :

n-port load, where $S = \text{Gamma0} * \text{eye}(\dots)$

`skrf.media.rectangularWaveguide.RectangularWaveguide.match`

`RectangularWaveguide.match` (*nports*=1, *z0*=None, ***kwargs*)

Perfect matched load ($\Gamma_0 = 0$).

Parameters `nports` : int

number of ports

`z0` : number, or array-like

characteristic impedance. Default is None, in which case the Media’s *z0* is used. This sets the resultant Network’s *z0*.

****kwargs** : key word arguments

passed to `Network` initializer

Returns `match` : `Network` object

a n-port match

Examples

```
>>> my_match = my_media.match(2, z0 = 50, name='Super Awesome Match')
```

skrf.media.rectangularWaveguide.RectangularWaveguide.open`RectangularWaveguide.open(nports=1, **kwargs)`Open ($\Gamma_0 = 1$)**Parameters** `nports` : int

number of ports

****kwargs** : key word argumentspassed 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 argumentspassed 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 argumentspassed 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.shuntRectangularWaveguide.**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 argumentspassed to `tee()`**Returns** *shunted_ntwk* : *Network* objecta shunted a *ntwk*. The resultant *shunted_ntwk* will have $(2 + \text{ntwk.number_of_ports} - 1)$ ports.**skrf.media.rectangularWaveguide.RectangularWaveguide.shunt_capacitor**RectangularWaveguide.**shunt_capacitor** (*C*, **args*, ***kwargs*)

Shunted capacitor

Parameters *C* : number, array-like

Capacitance in Farads.

args, **kwargs* : arguments, keyword argumentspassed to func:*delay_open*Returns** *shunt_capacitor* : *Network* object

shunted capacitor(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 argumentspassed 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 argumentspassed to `match()`, which is called initially to create a ‘blank’ network.**Returns** *thru* : `Network` object

matched transmission line of 0 length

See Also:`line` this just calls `line(0)`**skrf.media.rectangularWaveguide.RectangularWaveguide.white_gaussian_polar****RectangularWaveguide.white_gaussian_polar** (*phase_dev, mag_dev, n_ports=1, **kwargs*)

Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

Parameters *phase_mag* : number

standard deviation of magnitude

phase_dev : number

standard deviation of phase

n_ports : int

number of ports.

***kwargs* : passed to `Network`

initializer

Returns *result* : `Network` object

a noise network

skrf.media.rectangularWaveguide.RectangularWaveguide.write_csv**RectangularWaveguide.write_csv** (*filename='f, gamma, z0.csv'*)write this media’s frequency, *z0*, and *gamma* to a csv file.**Parameters** *filename* : string

file name to write out data to

See Also:`from_csv` class method to initialize Media object from a csv file written from this function**skrf.media.cpw.CPW****class** `skrf.media.cpw.CPW` (*frequency, w, s, ep_r, t=None, rho=None, *args, **kwargs*)

Coplanar waveguide class

This class was made from the technical documentation ²¹ provided by the qucs project ²² . The variables and properties of this class are coincident with their derivations.

Attributes

<code>K_ratio</code>	intermediary parameter. see qucs docs on cpw lines.
<code>alpha_conductor</code>	Losses due to conductor resistivity
<code>characteristic_impedance</code>	Characterisitic impedance
<code>ep_re</code>	intermediary parameter. see qucs docs on cpw lines.
<code>k1</code>	intermediary parameter. see qucs docs on cpw lines.
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

skrf.media.cpw.CPW.K_ratio

CPW.K_ratio

intermediary parameter. see qucs docs on cpw lines.

skrf.media.cpw.CPW.alpha_conductor

CPW.alpha_conductor

Losses due to conductor resistivity

Returns `alpha_conductor` : array-like

lossyness due to conductor losses

See Also :

————— :

surface_resistivity : calculates surface resistivity

skrf.media.cpw.CPW.characteristic_impedance

CPW.characteristic_impedance

Characterisitic impedance

The characteristic_impedance can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns `characteristic_impedance` : `numpy.ndarray`

skrf.media.cpw.CPW.ep_re

CPW.ep_re

intermediary parameter. see qucs docs on cpw lines.

skrf.media.cpw.CPW.k1

CPW.k1

intermediary parameter. see qucs docs on cpw lines.

²¹ <http://qucs.sourceforge.net/docs/technical.pdf>

²² <http://www.qucs.sourceforge.net/>

skrf.media.cpw.CPW.propagation_constant**CPW.propagation_constant**

Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the propagation constant to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns **propagation_constant** : `numpy.ndarray`
complex propagation constant for this media

Notes

propagation_constant must adhere to the following convention,

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

skrf.media.cpw.CPW.z0**CPW.z0**

Port Impedance

The port impedance is usually equal to the `characteristic_impedance`. Therefore, if the port impedance is `None` then this will return `characteristic_impedance`.

However, in some cases such as rectangular waveguide, the port impedance is traditionally set to 1 (normalized). In such a case this property may be used.

The Port Impedance can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns **port_impedance** : `numpy.ndarray`
the media's port impedance

Methods

<code>z0</code>	Characterisitic impedance
<code>__init__</code>	Coplanar Waveguide initializer
<code>capacitor</code>	Capacitor
<code>delay_load</code>	Delayed load
<code>delay_open</code>	Delayed open transmission line
<code>delay_short</code>	Delayed Short
<code>electrical_length</code>	calculates the electrical length for a given distance, at
<code>from_csv</code>	create a Media from numerical values stored in a csv file.
<code>gamma</code>	Propagation constant ..
<code>guess_length_of_delay_short</code>	Guess physical length of a delay short.
<code>impedance_mismatch</code>	Two-port network for an impedance miss-match
<code>inductor</code>	Inductor
<code>line</code>	Matched transmission line of given length
<code>load</code>	Load of given reflection coefficient.

Continued on next page

Table 3.47 – continued from previous page

<code>match</code>	Perfect matched load ($\Gamma_0 = 0$).
<code>open</code>	Open ($\Gamma_0 = 1$)
<code>resistor</code>	Resistor
<code>short</code>	Short ($\Gamma_0 = -1$)
<code>shunt</code>	Shunts a Network
<code>shunt_capacitor</code>	Shunted capacitor
<code>shunt_delay_load</code>	Shunted delayed load
<code>shunt_delay_open</code>	Shunted delayed open
<code>shunt_delay_short</code>	Shunted delayed short
<code>shunt_inductor</code>	Shunted inductor
<code>splitter</code>	Ideal, lossless n-way splitter.
<code>tee</code>	Ideal, lossless tee.
<code>theta_2_d</code>	Converts electrical length to physical distance.
<code>thru</code>	Matched transmission line of length 0.
<code>white_gaussian_polar</code>	Complex zero-mean gaussian white-noise network.
<code>write_csv</code>	write this media's frequency, <code>z0</code> , and gamma to a csv file.

skrf.media.cpw.CPW.Z0CPW.**Z0** ()

Characterisitic impedance

skrf.media.cpw.CPW.__init__CPW.**__init__** (*frequency*, *w*, *s*, *ep_r*, *t=None*, *rho=None*, **args*, ***kwargs*)

Coplanar Waveguide initializer

Parameters **frequency** : [Frequency](#) object

frequency band of this transmission line medium

w : number, or array-like

width of center conductor, in m.

s : number, or array-like

width of gap, in m.

ep_r : number, or array-like

relative permativity of substrate

t : number, or array-like, optional

conductor thickness, in m.

rho: number, or array-like, optional :

resistivity of conductor (None)

skrf.media.cpw.CPW.capacitorCPW.**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 argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **delay_open** : `Network` object

a delayed open

See Also:`delay_load` `delay_short` just calls this function**skrf.media.cpw.CPW.delay_short****CPW.delay_short** (*d*, *unit*='m', ***kwargs*)

Delayed Short

A transmission line of given length terminated with a short.

Parameters *d* : number

the length of transmissin line (see unit argument)

unit : ['m','deg','rad']**the units of d. possible options are:**

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

****kwargs** : key word argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **delay_short** : `Network` object

a delayed short

See Also:`delay_load` `delay_short` just calls this function

skrf.media.cpw.CPW.electrical_length**CPW.electrical_length** (*d*, *deg=False*)

calculates the electrical length for a given distance, at the center frequency.

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

delay distance, in meters

deg: Boolean :

return electal 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 argumentspassed to [`match\(\)`](#), which is called initially to create a 'blank' network.**Returns** **missmatch** : [`Network`](#) object

a 2-port network representing the impedance mismatch

See Also:

`match` called to create a ‘blank’ network

Notes

If `z1` and `z2` are arrays, they must be of same length as the `Media.frequency.npoints`

skrf.media.cpw.CPW.inductor

`CPW.inductor` (*L*, ***kwargs*)

Inductor

Parameters *L* : number, array

Inductance, in Henrys. If this is an array, must be of same length as frequency vector.

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns *inductor* : a 2-port `Network`

See Also:

`match` function called to create a ‘blank’ network

skrf.media.cpw.CPW.line

`CPW.line` (*d*, *unit='m'*, ***kwargs*)

Matched transmission line of given length

The units of *length* are interpreted according to the value of *unit*.

Parameters *d* : number

the length of transmissin line (see unit argument)

unit : ['m','deg','rad']

the units of *d*. possible options are:

- *m* : meters, physical length in meters (default)
- *deg* :degrees, electrical length in degrees
- *rad* :radians, electrical length in radians

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns *line* : `Network` object

matched tranmission line of given length

Examples

```
>>> my_media.line(90, 'deg', z0=50)
```

skrf.media.cpw.CPW.load**CPW.load** (*Gamma0*, *nports*=1, ***kwargs*)

Load of given reflection coefficient.

Parameters **Gamma0** : number, array-likeReflection coefficient of load (linear, not in db). If its an array it must be of shape: $k \times n$, where k is #frequency points in media, and n is *nports***nports** : int

number of ports

****kwargs** : key word argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **load** :class:'~skrf.network.Network' object :n-port load, where $S = \text{Gamma0} * \text{eye}(\dots)$ **skrf.media.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-likecharacteristic impedance. Default is None, in which case the Media's *z0* is used. This sets the resultant Network's *z0*.****kwargs** : key word argumentspassed 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 argumentspassed 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 $(2 + \text{ntwk.number_of_ports} - 1)$ 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 argumentspassed to func:*delay_openReturns** **shunt_capacitor** : *Network* object

shunted capacitor(2-port)

Notes

This calls:

```
shunt(capacitor(C, *args, **kwargs))
```

skrf.media.cpw.CPW.shunt_delay_load**CPW.shunt_delay_load** (*args, **kwargs)

Shunted delayed load

Parameters ***args, **kwargs** : arguments, keyword argumentspassed 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 argumentspassed to func:*delay_open***Returns** **shunt_delay_open** : *Network* object

shunted delayed open (2-port)

Notes

This calls:

```
shunt(delay_open(*args, **kwargs))
```

skrf.media.cpw.CPW.shunt_delay_short

CPW.**shunt_delay_short** (**args, **kwargs*)

Shunted delayed short

Parameters **args, **kwargs* : arguments, keyword arguments
passed to func:*delay_open*

Returns **shunt_delay_load** : *Network* object
shunted delayed open (2-port)

Notes

This calls:

```
shunt(delay_short(*args, **kwargs))
```

skrf.media.cpw.CPW.shunt_inductor

CPW.**shunt_inductor** (*L, *args, **kwargs*)

Shunted inductor

Parameters **L** : number, array-like
Inductance in Farads.
**args, **kwargs* : arguments, keyword arguments
passed to func:*delay_open*

Returns **shunt_inductor** : *Network* object
shunted inductor(2-port)

Notes

This calls:

```
shunt(inductor(C, *args, **kwargs))
```

skrf.media.cpw.CPW.splitter

CPW.**splitter** (*nports, **kwargs*)

Ideal, lossless n-way splitter.

Parameters **nports** : int
number of ports
***kwargs* : key word arguments
passed to `match()`, which is called initially to create a ‘blank’ network.

Returns **tee** : *Network* object
a n-port splitter

See Also:

`match` called to create a ‘blank’ network

skrf.media.cpw.CPW.tee

`CPW.tee` (***kwargs*)

Ideal, lossless tee. (3-port splitter)

Parameters ***kwargs* : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `tee` : `Network` object

a 3-port splitter

See Also:

`splitter` this just calls `splitter(3)`

`match` called to create a ‘blank’ network

skrf.media.cpw.CPW.theta_2_d

`CPW.theta_2_d` (*theta, deg=True*)

Converts electrical length to physical distance.

The given electrical length is to be at the center frequency.

Parameters *theta* : number

electrical length, at band center (see *deg* for unit)

deg : Boolean

is *theta* in degrees?

Returns *d* : number

physical distance in meters

skrf.media.cpw.CPW.thru

`CPW.thru` (***kwargs*)

Matched transmission line of length 0.

Parameters ***kwargs* : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `thru` : `Network` object

matched transmission line of 0 length

See Also:

`line` this just calls `line(0)`

skrf.media.cpw.CPW.white_gaussian_polarCPW.**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_csvCPW.**write_csv** (*filename='f, gamma, z0.csv'*)

write this media’s frequency, z0, and gamma to a csv file.

Parameters **filename** : string

file name to write out data to

See Also:[from_csv](#) class method to initialize Media object from a csv file written from this function**skrf.media.freespace.Freespace****class** [skrf.media.freespace.Freespace](#) (*frequency, ep_r=1, mu_r=1, *args, **kwargs*)

Represents a plane-wave in a homogeneous freespace, defined by the space’s relative permativity and relative permeability.

The field properties of space are related to a distributed circuit transmission line model given in circuit theory by:

Circuit Property	Field Property
distributed_capacitance	real($\epsilon_p_0 \epsilon_p_r$)
distributed_resistance	imag($\epsilon_p_0 \epsilon_p_r$)
distributed_inductance	real($\mu_0 \mu_r$)
distributed_conductance	imag($\mu_0 \mu_r$)

This class’s inheritance is; [Media](#)->[DistributedCircuit](#)->[Freespace](#)

Attributes

<code>Y</code>	Distributed Admittance, Y'
<code>Z</code>	Distributed Impedance, Z'
<code>characteristic_impedance</code>	Characterisitic impedance
<code>propagation_constant</code>	Propagation constant
<code>z0</code>	Port Impedance

skrf.media.freespace.Freespace.Y`Freespace.Y`Distributed Admittance, Y'

Defined as

$$Y' = G' + j\omega C'$$

Returns `Y` : `numpy.ndarray`

Distributed Admittance in units of S/m

skrf.media.freespace.Freespace.Z`Freespace.Z`Distributed Impedance, Z'

Defined as

$$Z' = R' + j\omega I'$$

Returns `Z` : `numpy.ndarray`

Distributed impedance in units of ohm/m

skrf.media.freespace.Freespace.characteristic_impedance`Freespace.characteristic_impedance`

Characterisitic impedance

The characteristic_impedance can be either a number, array-like, or a function. If it is a function is must take no arguments. The reason to make it a function is if you want the characterisitic impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns `characteristic_impedance` : `numpy.ndarray`**skrf.media.freespace.Freespace.propagation_constant**`Freespace.propagation_constant`

Propagation constant

The propagation constant can be either a number, array-like, or a function. If it is a function 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 it must take no arguments. The reason to make it a function is if you want the Port Impedance to be dynamic, meaning changing with some attribute of the media. See `__init__()` for more explanation.

Returns `port_impedance` : `numpy.ndarray`
the media's port impedance

Methods

<code>z0</code>	Characteristic Impedance, Z_0
<code>__init__</code>	Freespace initializer
<code>capacitor</code>	Capacitor
<code>delay_load</code>	Delayed load
<code>delay_open</code>	Delayed open transmission line
<code>delay_short</code>	Delayed Short
<code>electrical_length</code>	calculates the electrical length for a given distance, at
<code>from_Media</code>	Initializes a DistributedCircuit from an existing
<code>from_csv</code>	create a Media from numerical values stored in a csv file.
<code>gamma</code>	Propagation Constant, γ
<code>guess_length_of_delay_short</code>	Guess physical length of a delay short.
<code>impedance_mismatch</code>	Two-port network for an impedance miss-match
<code>inductor</code>	Inductor
<code>line</code>	Matched transmission line of given length
<code>load</code>	Load of given reflection coefficient.
<code>match</code>	Perfect matched load ($\Gamma_0 = 0$).
<code>open</code>	Open ($\Gamma_0 = 1$)
<code>resistor</code>	Resistor
<code>short</code>	Short ($\Gamma_0 = -1$)
<code>shunt</code>	Shunts a <code>Network</code>
<code>shunt_capacitor</code>	Shunted capacitor
<code>shunt_delay_load</code>	Shunted delayed load
<code>shunt_delay_open</code>	Shunted delayed open
<code>shunt_delay_short</code>	Shunted delayed short
<code>shunt_inductor</code>	Shunted inductor

Continued on next page

Table 3.49 – continued from previous page

<code>splitter</code>	Ideal, lossless n-way splitter.
<code>tee</code>	Ideal, lossless tee.
<code>theta_2_d</code>	Converts electrical length to physical distance.
<code>thru</code>	Matched transmission line of length 0.
<code>white_gaussian_polar</code>	Complex zero-mean gaussian white-noise network.
<code>write_csv</code>	write this media's frequency, z0, and gamma to a csv file.

skrf.media.freespace.Freespace.Z0`Freespace.Z0()`Characteristic Impedance, Z_0

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

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
<code>distributed_capacitance</code>	<code>real(ep_0*ep_r)</code>
<code>distributed_resistance</code>	<code>imag(ep_0*ep_r)</code>
<code>distributed_inductance</code>	<code>real(mu_0*mu_r)</code>
<code>distributed_conductance</code>	<code>imag(mu_0*mu_r)</code>

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 argumentspassed 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 argumentspassed to `match()`, which is called initially to create a 'blank' network.**Returns** **delay_short** : `Network` object

a delayed short

See Also:`delay_load` `delay_short` just calls this function

skrf.media.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 electal 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 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 argumentspassed to `match()`, which is called initially to create a ‘blank’ network.**Returns** **mismatch** : `Network` object

a 2-port network representing the impedance mismatch

See Also:`match` called to create a ‘blank’ network**Notes**If *z1* and *z2* are arrays, they must be of same length as the `Media.frequency.npoints`**skrf.media.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 argumentspassed 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 transmission line of given length

Examples

```
>>> my_media.line(90, 'deg', z0=50)
```

skrf.media.freespace.Freespace.load

`Freespace.load` (*Gamma0*, *nports*=1, ****kwargs**)

Load of given reflection coefficient.

Parameters `Gamma0` : number, array-like

Reflection coefficient of load (linear, not in db). If its an array it must be of shape: $k \times n$, where k is #frequency points in media, and n is *nports*

nports : int

number of ports

****kwargs** : key word arguments

passed to `match()`, which is called initially to create a ‘blank’ network.

Returns `load` :class:‘~skrf.network.Network’ object :

n -port load, where $S = \text{Gamma0} * \text{eye}(\dots)$

skrf.media.freespace.Freespace.match

`Freespace.match` (*nports*=1, *z0*=None, ****kwargs**)

Perfect matched load ($\Gamma_0 = 0$).

Parameters **nports** : int

number of ports

z0 : number, or array-like

characteristic impedance. Default is None, in which case the Media’s *z0* is used. This sets the resultant Network’s *z0*.

****kwargs** : key word arguments

passed to `Network` initializer

Returns `match` : `Network` object

a n -port match

Examples

```
>>> my_match = my_media.match(2, z0 = 50, name='Super Awesome Match')
```

skrf.media.freespace.Freespace.open**Freespace.open** (*nports=1*, ***kwargs*)Open ($\Gamma_0 = 1$)**Parameters** **nports** : int

number of ports

****kwargs** : key word argumentspassed 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 argumentspassed 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 argumentspassed 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 argumentspassed to *tee()***Returns** *shunted_ntwk* : *Network* objecta shunted a *ntwk*. The resultant *shunted_ntwk* will have $(2 + \text{ntwk.number_of_ports} - 1)$ ports.**skrf.media.freespace.Freespace.shunt_capacitor****Freespace.shunt_capacitor** (*C*, **args*, ***kwargs*)

Shunted capacitor

Parameters *C* : number, array-like

Capacitance in Farads.

args, **kwargs* : arguments, keyword argumentspassed to func:*delay_open*Returns** *shunt_capacitor* : *Network* object

shunted capacitor(2-port)

Notes

This calls:

```
shunt(capacitor(C,*args, **kwargs))
```

skrf.media.freespace.Freespace.shunt_delay_load**Freespace.shunt_delay_load** (**args*, ***kwargs*)

Shunted delayed load

Parameters **args, **kwargs* : arguments, keyword argumentspassed 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 argumentspassed to `match()`, which is called initially to create a ‘blank’ network.**Returns** **thru** : `Network` object

matched transmission line of 0 length

See Also:`line` this just calls `line(0)`**skrf.media.freespace.Freespace.white_gaussian_polar****Freespace.white_gaussian_polar** (phase_dev, mag_dev, n_ports=1, **kwargs)

Complex zero-mean gaussian white-noise network.

Creates a network whose s-matrix is complex zero-mean gaussian white-noise, of given standard deviations for phase and magnitude components. This ‘noise’ network can be added to networks to simulate additive noise.

Parameters **phase_mag** : number

standard deviation of magnitude

phase_dev : number

standard deviation of phase

n_ports : int

number of ports.

****kwargs** : passed to `Network`

initializer

Returns **result** : `Network` object

a noise network

skrf.media.freespace.Freespace.write_csv**Freespace.write_csv** (filename='f, gamma, z0.csv')

write this media’s frequency, z0, and gamma to a csv file.

Parameters **filename** : string

file name to write out data to

See Also:`from_csv` class method to initialize Media object from a csv file written from this function

3.3.3 Indices and tables

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

BIBLIOGRAPHY

- [R5] 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>
- [R6] 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>

PYTHON MODULE INDEX

S

- `skrf.calibration`, 90
- `skrf.calibration.calibration`, 90
- `skrf.calibration.calibrationAlgorithms`,
97
- `skrf.calibration.calibrationFunctions`,
101
- `skrf.convenience`, 74
- `skrf.frequency`, 65
- `skrf.mathFunctions`, 79
- `skrf.media`, 108
- `skrf.network`, 35
- `skrf.networkSet`, 60
- `skrf.plotting`, 69
- `skrf.tlineFunctions`, 81
- `skrf.touchstone`, 73

PYTHON MODULE INDEX

S

- `skrf.calibration`, 90
- `skrf.calibration.calibration`, 90
- `skrf.calibration.calibrationAlgorithms`,
97
- `skrf.calibration.calibrationFunctions`,
101
- `skrf.convenience`, 74
- `skrf.frequency`, 65
- `skrf.mathFunctions`, 79
- `skrf.media`, 108
- `skrf.network`, 35
- `skrf.networkSet`, 60
- `skrf.plotting`, 69
- `skrf.tlineFunctions`, 81
- `skrf.touchstone`, 73

INDEX

Symbols

- `__init__()` (skrf.calibration.calibration.Calibration method), 93, 104
 - `__init__()` (skrf.frequency.Frequency method), 67
 - `__init__()` (skrf.media.cpw.CPW method), 151
 - `__init__()` (skrf.media.distributedCircuit.DistributedCircuit method), 124
 - `__init__()` (skrf.media.freespace.Freespace method), 164
 - `__init__()` (skrf.media.media.Media method), 111
 - `__init__()` (skrf.media.rectangularWaveguide.RectangularWaveguide method), 138
 - `__init__()` (skrf.network.Network method), 39
 - `__init__()` (skrf.networkSet.NetworkSet method), 62
 - `__init__()` (skrf.touchstone.touchstone method), 73
- ## A
- `abc_2_coefs_dict()` (in module skrf.calibration.calibrationAlgorithms), 101
 - `add_markers_to_lines()` (in module skrf.convenience), 75
 - `add_noise_polar()` (skrf.network.Network method), 40
 - `add_noise_polar_flatband()` (skrf.network.Network method), 40
 - `alpha_conductor` (skrf.media.cpw.CPW attribute), 149
 - `apply_cal()` (skrf.calibration.calibration.Calibration method), 94, 105
 - `apply_cal_to_all_in_dir()` (skrf.calibration.calibration.Calibration method), 94, 105
 - `average()` (in module skrf.network), 58
- ## B
- `biased_error()` (skrf.calibration.calibration.Calibration method), 94, 106
- ## C
- `caled_ntwk_sets` (skrf.calibration.calibration.Calibration attribute), 91, 102
 - `caled_ntwks` (skrf.calibration.calibration.Calibration attribute), 91, 102
 - `Calibration` (class in skrf.calibration.calibration), 90, 102
 - `calibration_algorithm_dict` (skrf.calibration.calibration.Calibration attribute), 91, 102
 - `capacitor()` (skrf.media.cpw.CPW method), 151
 - `capacitor()` (skrf.media.distributedCircuit.DistributedCircuit method), 124
 - `capacitor()` (skrf.media.freespace.Freespace method), 164
 - `capacitor()` (skrf.media.media.Media method), 111
 - `capacitor()` (skrf.media.rectangularWaveguide.RectangularWaveguide method), 138
 - `cartesian_product_calibration_set()` (in module skrf.calibration.calibrationFunctions), 101
 - `cascade()` (in module skrf.network), 45
 - `center` (skrf.frequency.Frequency attribute), 65
 - `characteristic_impedance` (skrf.media.cpw.CPW attribute), 149
 - `characteristic_impedance` (skrf.media.distributedCircuit.DistributedCircuit attribute), 122
 - `characteristic_impedance` (skrf.media.freespace.Freespace attribute), 162
 - `characteristic_impedance` (skrf.media.media.Media attribute), 109
 - `characteristic_impedance` (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 135
 - `coefs` (skrf.calibration.calibration.Calibration attribute), 91, 103
 - `complex2Scalar()` (in module skrf.mathFunctions), 81
 - `complex_2_db()` (in module skrf.mathFunctions), 79
 - `complex_2_degree()` (in module skrf.mathFunctions), 79
 - `complex_2_magnitude()` (in module skrf.mathFunctions), 79, 80
 - `complex_2_radian()` (in module skrf.mathFunctions), 79
 - `complex_2_reim()` (in module skrf.mathFunctions), 79
 - `connect()` (in module skrf.network), 44
 - `connect_s()` (in module skrf.network), 48
 - `copy()` (skrf.frequency.Frequency method), 68
 - `copy()` (skrf.network.Network method), 40
 - `copy()` (skrf.networkSet.NetworkSet method), 62
 - `CPW` (class in skrf.media.cpw), 148

csv_2_touchstone() (in module skrf.network), 60

D

db_2_np() (in module skrf.mathFunctions), 81

de_embed() (in module skrf.network), 46

degree_2_radian() (in module skrf.mathFunctions), 80

delay_load() (skrf.media.cpw.CPW method), 152

delay_load() (skrf.media.distributedCircuit.DistributedCircuit method), 125

delay_load() (skrf.media.freespace.Freespace method), 165

delay_load() (skrf.media.media.Media method), 112

delay_load() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 139

delay_open() (skrf.media.cpw.CPW method), 153

delay_open() (skrf.media.distributedCircuit.DistributedCircuit method), 126

delay_open() (skrf.media.freespace.Freespace method), 166

delay_open() (skrf.media.media.Media method), 112

delay_open() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 140

delay_short() (skrf.media.cpw.CPW method), 153

delay_short() (skrf.media.distributedCircuit.DistributedCircuit method), 126

delay_short() (skrf.media.freespace.Freespace method), 166

delay_short() (skrf.media.media.Media method), 113

delay_short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 140

dirac_delta() (in module skrf.mathFunctions), 81

distance_2_electrical_length() (in module skrf.tlineFunctions), 85

distributed_circuit_2_propagation_impedance() (in module skrf.tlineFunctions), 88

DistributedCircuit (class in skrf.media.distributedCircuit), 121

E

eight_term_2_one_port_coefs() (in module skrf.calibration.calibrationAlgorithms), 101

electrical_length() (skrf.media.cpw.CPW method), 154

electrical_length() (skrf.media.distributedCircuit.DistributedCircuit method), 127

electrical_length() (skrf.media.freespace.Freespace method), 167

electrical_length() (skrf.media.media.Media method), 113

electrical_length() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 141

electrical_length_2_distance() (in module skrf.tlineFunctions), 85

element_wise_method() (skrf.networkSet.NetworkSet method), 62

ep (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 135

ep_re (skrf.media.cpw.CPW attribute), 149

error_ntwk (skrf.calibration.calibration.Calibration attribute), 91, 103

F

f (skrf.frequency.Frequency attribute), 66

f (skrf.network.Network attribute), 37

f_scaled (skrf.frequency.Frequency attribute), 66

find_nearest() (in module skrf.convenience), 77

find_nearest_index() (in module skrf.convenience), 77

freespace (skrf.media.freespace.Freespace method), 40

Freespace (class in skrf.media.freespace), 161

Frequency (class in skrf.frequency), 65

frequency (skrf.network.Network attribute), 37

from_csv() (skrf.media.cpw.CPW class method), 154

from_csv() (skrf.media.distributedCircuit.DistributedCircuit class method), 127

from_csv() (skrf.media.freespace.Freespace class method), 167

from_csv() (skrf.media.media.Media class method), 114

from_csv() (skrf.media.rectangularWaveguide.RectangularWaveguide class method), 141

from_f() (skrf.frequency.Frequency class method), 68

from_Media() (skrf.media.distributedCircuit.DistributedCircuit class method), 127

from_Media() (skrf.media.freespace.Freespace class method), 167

from_zip() (skrf.networkSet.NetworkSet class method), 62

func_on_all_figs() (in module skrf.convenience), 75

func_per_standard() (skrf.calibration.calibration.Calibration method), 94, 106

G

gamma() (skrf.media.cpw.CPW method), 154

gamma() (skrf.media.distributedCircuit.DistributedCircuit method), 127

gamma() (skrf.media.freespace.Freespace method), 167

Gamma0_2_Gamma_in() (in module skrf.tlineFunctions), 84

Gamma0_2_zin() (in module skrf.tlineFunctions), 84

Gamma0_2_zl() (in module skrf.tlineFunctions), 84

get_format() (skrf.touchstone.touchstone method), 74

get_noise_data() (skrf.touchstone.touchstone method), 74

get_noise_names() (skrf.touchstone.touchstone method), 74

get_arrays() (skrf.touchstone.touchstone method), 74

get_sparameter_data() (skrf.touchstone.touchstone method), 74

get_sparameter_names() (skrf.touchstone.touchstone method), 74

- guess_length_of_delay_short() (skrf.media.cpw.CPW method), 154
- guess_length_of_delay_short() (skrf.media.distributedCircuit.DistributedCircuit method), 127
- guess_length_of_delay_short() (skrf.media.freespace.Freespace method), 167
- guess_length_of_delay_short() (skrf.media.media.Media method), 114
- guess_length_of_delay_short() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 141
- ## H
- hfss_touchstone_2_gamma_z0() (in module skrf.convenience), 76
- hfss_touchstone_2_media() (in module skrf.convenience), 76
- ## I
- impedance_mismatch() (in module skrf.network), 59
- impedance_mismatch() (skrf.media.cpw.CPW method), 154
- impedance_mismatch() (skrf.media.distributedCircuit.DistributedCircuit method), 128
- impedance_mismatch() (skrf.media.freespace.Freespace method), 168
- impedance_mismatch() (skrf.media.media.Media method), 114
- impedance_mismatch() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 141
- inductor() (skrf.media.cpw.CPW method), 155
- inductor() (skrf.media.distributedCircuit.DistributedCircuit method), 128
- inductor() (skrf.media.freespace.Freespace method), 168
- inductor() (skrf.media.media.Media method), 114
- inductor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 142
- innerconnect() (in module skrf.network), 45
- innerconnect_s() (in module skrf.network), 49
- input_impedance_at_theta() (in module skrf.tlineFunctions), 87
- interpolate() (skrf.network.Network method), 40, 46
- interpolate_self() (skrf.network.Network method), 41, 47
- interpolate_self_npoints() (skrf.network.Network method), 41, 47
- inv (skrf.network.Network attribute), 37
- inv (skrf.networkSet.NetworkSet attribute), 61
- inv() (in module skrf.network), 48
- ## K
- k0 (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 135
- k1 (skrf.media.cpw.CPW attribute), 149
- K_ratio (skrf.media.cpw.CPW attribute), 149
- kc (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 135
- kx (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 136
- ky (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 136
- kz() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 142
- ## L
- labelXAxis() (skrf.frequency.Frequency method), 69
- legend_off() (in module skrf.convenience), 75
- line() (skrf.media.cpw.CPW method), 155
- line() (skrf.media.distributedCircuit.DistributedCircuit method), 128
- line() (skrf.media.freespace.Freespace method), 168
- line() (skrf.media.media.Media method), 115
- line() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 142
- load() (skrf.media.cpw.CPW method), 156
- load() (skrf.media.distributedCircuit.DistributedCircuit method), 129
- load() (skrf.media.freespace.Freespace method), 169
- load() (skrf.media.media.Media method), 115
- load() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 143
- load_all_touchstones() (in module skrf.network), 59
- load_file() (skrf.touchstone.touchstone method), 74
- load_impedance_2_reflection_coefficient() (in module skrf.tlineFunctions), 87
- load_impedance_2_reflection_coefficient_at_theta() (in module skrf.tlineFunctions), 88
- ## M
- match() (skrf.media.cpw.CPW method), 156
- match() (skrf.media.distributedCircuit.DistributedCircuit method), 129
- match() (skrf.media.freespace.Freespace method), 169
- match() (skrf.media.media.Media method), 116
- match() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 143
- mean_residuals() (skrf.calibration.calibration.Calibration method), 94, 106
- mean_s_db (skrf.networkSet.NetworkSet attribute), 61
- Media (class in skrf.media.media), 109
- mu (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 136
- multiplier (skrf.frequency.Frequency attribute), 66
- multiplier_dict (skrf.frequency.Frequency attribute), 66
- multiply_noise() (skrf.network.Network method), 41

N

Network (class in skrf.network), 35
NetworkSet (class in skrf.networkSet), 60
neuman() (in module skrf.mathFunctions), 81
now_string() (in module skrf.convenience), 77
np_2_db() (in module skrf.mathFunctions), 80
npoints (skrf.frequency.Frequency attribute), 66
nports (skrf.calibration.calibration.Calibration attribute), 91, 103
nports (skrf.network.Network attribute), 37
nstandards (skrf.calibration.calibration.Calibration attribute), 91, 103
nudge() (skrf.network.Network method), 42
null() (in module skrf.mathFunctions), 81
number_of_ports (skrf.network.Network attribute), 38

O

one_port() (in module skrf.calibration.calibrationAlgorithms), 97
one_port_2_two_port() (in module skrf.network), 58
one_port_nls() (in module skrf.calibration.calibrationAlgorithms), 98
open() (skrf.media.cpw.CPW method), 156
open() (skrf.media.distributedCircuit.DistributedCircuit method), 130
open() (skrf.media.freespace.Freespace method), 170
open() (skrf.media.media.Media method), 116
open() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 144
output_from_cal (skrf.calibration.calibration.Calibration attribute), 92, 103

P

parameterized_self_calibration() (in module skrf.calibration.calibrationAlgorithms), 99
parameterized_self_calibration_nls() (in module skrf.calibration.calibrationAlgorithms), 100
passivity (skrf.network.Network attribute), 38
plot_coefs_db() (skrf.calibration.calibration.Calibration method), 95, 106
plot_complex_polar() (in module skrf.plotting), 72
plot_complex_rectangular() (in module skrf.plotting), 72
plot_errors() (skrf.calibration.calibration.Calibration method), 95, 106
plot_it_all() (skrf.network.Network method), 42
plot_logsigma() (skrf.networkSet.NetworkSet method), 63
plot_passivity() (skrf.network.Network method), 42
plot_polar() (in module skrf.plotting), 71
plot_rectangular() (in module skrf.plotting), 70
plot_residuals() (skrf.calibration.calibration.Calibration method), 95, 106
plot_residuals_db() (skrf.calibration.calibration.Calibration method), 95, 106

plot_residuals_mag() (skrf.calibration.calibration.Calibration method), 95, 107
plot_residuals_smith() (skrf.calibration.calibration.Calibration method), 95, 107
plot_s_smith() (skrf.network.Network method), 42
plot_smith() (in module skrf.plotting), 69
plot_uncertainty_bounds_component() (skrf.networkSet.NetworkSet method), 63
plot_uncertainty_bounds_s() (skrf.networkSet.NetworkSet method), 63
plot_uncertainty_bounds_s_db() (skrf.networkSet.NetworkSet method), 64
plot_uncertainty_decomposition() (skrf.networkSet.NetworkSet method), 64
plot_uncertainty_per_standard() (skrf.calibration.calibration.Calibration method), 95, 107
propagation_constant (skrf.media.cpw.CPW attribute), 150
propagation_constant (skrf.media.distributedCircuit.DistributedCircuit attribute), 122
propagation_constant (skrf.media.freespace.Freespace attribute), 162
propagation_constant (skrf.media.media.Media attribute), 109
propagation_constant (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 136
propagation_impedance_2_distributed_circuit() (in module skrf.tlineFunctions), 89

R

radian_2_degree() (in module skrf.mathFunctions), 80
read_touchstone() (skrf.network.Network method), 43
RectangularWaveguide (class in skrf.media.rectangularWaveguide), 134
reflection_coefficient_2_input_impedance() (in module skrf.tlineFunctions), 86
reflection_coefficient_2_input_impedance_at_theta() (in module skrf.tlineFunctions), 87
reflection_coefficient_at_theta() (in module skrf.tlineFunctions), 86
renumber() (skrf.network.Network method), 43
residual_ntwks (skrf.calibration.calibration.Calibration attribute), 92, 103
residuals (skrf.calibration.calibration.Calibration attribute), 92, 103
resistor() (skrf.media.cpw.CPW method), 157
resistor() (skrf.media.distributedCircuit.DistributedCircuit method), 130
resistor() (skrf.media.freespace.Freespace method), 170
resistor() (skrf.media.media.Media method), 116
resistor() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 144

`run()` (`skrf.calibration.calibration.Calibration` method), 96, 107

S

`s` (`skrf.network.Network` attribute), 38

`s2t()` (in module `skrf.network`), 51

`s2y()` (in module `skrf.network`), 50

`s2z()` (in module `skrf.network`), 50

`save_all_figs()` (in module `skrf.convenience`), 75

`scalar2Complex()` (in module `skrf.mathFunctions`), 81

`set_wise_function()` (`skrf.networkSet.NetworkSet` method), 64

`short()` (`skrf.media.cpw.CPW` method), 157

`short()` (`skrf.media.distributedCircuit.DistributedCircuit` method), 130

`short()` (`skrf.media.freespace.Freespace` method), 170

`short()` (`skrf.media.media.Media` method), 117

`short()` (`skrf.media.rectangularWaveguide.RectangularWaveguide` method), 144

`shunt()` (`skrf.media.cpw.CPW` method), 157

`shunt()` (`skrf.media.distributedCircuit.DistributedCircuit` method), 131

`shunt()` (`skrf.media.freespace.Freespace` method), 171

`shunt()` (`skrf.media.media.Media` method), 117

`shunt()` (`skrf.media.rectangularWaveguide.RectangularWaveguide` method), 145

`shunt_capacitor()` (`skrf.media.cpw.CPW` method), 158

`shunt_capacitor()` (`skrf.media.distributedCircuit.DistributedCircuit` method), 131

`shunt_capacitor()` (`skrf.media.freespace.Freespace` method), 171

`shunt_capacitor()` (`skrf.media.media.Media` method), 117

`shunt_capacitor()` (`skrf.media.rectangularWaveguide.RectangularWaveguide` method), 145

`shunt_delay_load()` (`skrf.media.cpw.CPW` method), 158

`shunt_delay_load()` (`skrf.media.distributedCircuit.DistributedCircuit` method), 131

`shunt_delay_load()` (`skrf.media.freespace.Freespace` method), 171

`shunt_delay_load()` (`skrf.media.media.Media` method), 118

`shunt_delay_load()` (`skrf.media.rectangularWaveguide.RectangularWaveguide` method), 145

`shunt_delay_open()` (`skrf.media.cpw.CPW` method), 158

`shunt_delay_open()` (`skrf.media.distributedCircuit.DistributedCircuit` method), 132

`shunt_delay_open()` (`skrf.media.freespace.Freespace` method), 172

`shunt_delay_open()` (`skrf.media.media.Media` method), 118

`shunt_delay_open()` (`skrf.media.rectangularWaveguide.RectangularWaveguide` method), 146

`shunt_delay_short()` (`skrf.media.cpw.CPW` method), 159

`shunt_delay_short()` (`skrf.media.distributedCircuit.DistributedCircuit` method), 132

`shunt_delay_short()` (`skrf.media.freespace.Freespace` method), 172

`shunt_delay_short()` (`skrf.media.media.Media` method), 118

`shunt_delay_short()` (`skrf.media.rectangularWaveguide.RectangularWaveguide` method), 146

`shunt_inductor()` (`skrf.media.cpw.CPW` method), 159

`shunt_inductor()` (`skrf.media.distributedCircuit.DistributedCircuit` method), 132

`shunt_inductor()` (`skrf.media.freespace.Freespace` method), 172

`shunt_inductor()` (`skrf.media.media.Media` method), 119

`shunt_inductor()` (`skrf.media.rectangularWaveguide.RectangularWaveguide` method), 146

`signature()` (`skrf.networkSet.NetworkSet` method), 64

`skin_depth()` (in module `skrf.tlineFunctions`), 89

`skrf.calibration` (module), 90

`skrf.calibration.calibration` (module), 90

`skrf.calibration.calibrationAlgorithms` (module), 97

`skrf.calibration.calibrationFunctions` (module), 101

`skrf.convenience` (module), 74

`skrf.frequency` (module), 65

`skrf.mathFunctions` (module), 79

`skrf.media` (module), 108

`skrf.network` (module), 35

`skrf.networkSet` (module), 60

`skrf.plotting` (module), 69

`skrf.tlineFunctions` (module), 81

`skrf.touchstone` (module), 73

`smith()` (in module `skrf.plotting`), 69

`span` (`skrf.frequency.Frequency` attribute), 66

`splitter()` (`skrf.media.cpw.CPW` method), 159

`splitter()` (`skrf.media.distributedCircuit.DistributedCircuit` method), 133

`splitter()` (`skrf.media.freespace.Freespace` method), 173

`splitter()` (`skrf.media.media.Media` method), 119

`splitter()` (`skrf.media.rectangularWaveguide.RectangularWaveguide` method), 147

`sqrt_phase_unwrap()` (in module `skrf.mathFunctions`), 80

`start` (`skrf.frequency.Frequency` attribute), 67

`std_s_db` (`skrf.networkSet.NetworkSet` attribute), 61

`step` (`skrf.frequency.Frequency` attribute), 67

`stop` (`skrf.frequency.Frequency` attribute), 67

`surface_resistivity()` (in module `skrf.tlineFunctions`), 90

T

`t` (`skrf.network.Network` attribute), 38

`t2s()` (in module `skrf.network`), 56

`t2y()` (in module `skrf.network`), 57

`t2z()` (in module `skrf.network`), 56

`tee()` (`skrf.media.cpw.CPW` method), 160

tee() (skrf.media.distributedCircuit.DistributedCircuit method), 133

tee() (skrf.media.freespace.Freespace method), 173

tee() (skrf.media.media.Media method), 119

tee() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 147

theta() (in module skrf.tlineFunctions), 82

theta_2_d() (skrf.media.cpw.CPW method), 160

theta_2_d() (skrf.media.distributedCircuit.DistributedCircuit method), 133

theta_2_d() (skrf.media.freespace.Freespace method), 173

theta_2_d() (skrf.media.media.Media method), 120

theta_2_d() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 147

thru() (skrf.media.cpw.CPW method), 160

thru() (skrf.media.distributedCircuit.DistributedCircuit method), 134

thru() (skrf.media.freespace.Freespace method), 174

thru() (skrf.media.media.Media method), 120

thru() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 148

total_error() (skrf.calibration.calibration.Calibration method), 96, 107

touchstone (class in skrf.touchstone), 73

Ts (skrf.calibration.calibration.Calibration attribute), 91, 102

two_port() (in module skrf.calibration.calibrationAlgorithms), 98

type (skrf.calibration.calibration.Calibration attribute), 92, 104

U

unbiased_error() (skrf.calibration.calibration.Calibration method), 96, 108

uncertainty_ntwk_triplet() (skrf.networkSet.NetworkSet method), 65

uncertainty_per_standard() (skrf.calibration.calibration.Calibration method), 96, 108

unit (skrf.frequency.Frequency attribute), 67

unit_dict (skrf.frequency.Frequency attribute), 67

unterminate_switch_terms() (in module skrf.calibration.calibrationAlgorithms), 100

unwrap_rad() (in module skrf.mathFunctions), 80

W

w (skrf.frequency.Frequency attribute), 67

white_gaussian_polar() (skrf.media.cpw.CPW method), 161

white_gaussian_polar() (skrf.media.distributedCircuit.DistributedCircuit method), 134

white_gaussian_polar() (skrf.media.freespace.Freespace method), 174

white_gaussian_polar() (skrf.media.media.Media method), 120

white_gaussian_polar() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 148

write_csv() (skrf.media.cpw.CPW method), 161

write_csv() (skrf.media.distributedCircuit.DistributedCircuit method), 134

write_csv() (skrf.media.freespace.Freespace method), 174

write_csv() (skrf.media.media.Media method), 121

write_csv() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 148

write_dict_of_networks() (in module skrf.network), 59

Writeguide() (skrf.network.Network method), 43

Y

Y (skrf.media.distributedCircuit.DistributedCircuit attribute), 122

Y (skrf.media.freespace.Freespace attribute), 162

y (skrf.network.Network attribute), 39

y2s() (in module skrf.network), 54

y2t() (in module skrf.network), 55

y2z() (in module skrf.network), 55

Z

Z (skrf.media.distributedCircuit.DistributedCircuit attribute), 122

Z (skrf.media.freespace.Freespace attribute), 162

z (skrf.network.Network attribute), 39

z0 (skrf.media.cpw.CPW attribute), 150

z0 (skrf.media.distributedCircuit.DistributedCircuit attribute), 123

z0 (skrf.media.freespace.Freespace attribute), 163

z0 (skrf.media.media.Media attribute), 110

z0 (skrf.media.rectangularWaveguide.RectangularWaveguide attribute), 137

z0 (skrf.network.Network attribute), 39

Z0() (skrf.media.cpw.CPW method), 151

Z0() (skrf.media.distributedCircuit.DistributedCircuit method), 124

Z0() (skrf.media.freespace.Freespace method), 164

Z0() (skrf.media.rectangularWaveguide.RectangularWaveguide method), 137

z2s() (in module skrf.network), 52

z2t() (in module skrf.network), 53

z2y() (in module skrf.network), 52

zl_2_Gamma0() (in module skrf.tlineFunctions), 83

zl_2_Gamma_in() (in module skrf.tlineFunctions), 84

zl_2_zin() (in module skrf.tlineFunctions), 84