mwavepy Documentation

Release 1.3

alex arsenovic

August 21, 2011

CONTENTS

Contents:

CHAPTER

ONE

INSTALLATION

1.1 Requirements

The requirements are basically a python environment setup to do numerical/scientific computing. If you are new to Python development, I recommend you 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, there is a list of requirements at end of this section.

NOTE: if you want to use mwavepy for instrument control you will need to install pyvisa manually. The link is given in List of Requirements section. Also, you may be interested in David Urso's Pythics module, for easy gui creation.

1.2 Install mwavepy

There are three choices for installing mwavepy:

- windows installer
- python source package
- SVN version

They can all be found here http://code.google.com/p/mwavepy/downloads/list

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

If you know how to install a python package but aren't familiar with SVN then you want the Python source package . Examples, documentation, and installation instructions are provided in the the python package.

If you know how to use SVN, I recommend the SVN version because it has more features.

1.3 Linux-Specific

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
```

1.4 List of Requirements

Here is a list of the requirements, 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:

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

CHAPTER

TWO

QUICK INTRODUCTION

This quick intro is aimed at those who are familiar with python, or are impatient. If you want a slower introduction, see the *Slow Introduction*.

2.1 Loading Touchstone Files

First, import mwavepy and name it something short, like ''mv'':

```
import mwavepy as mv
```

Create a few *Network's* from touchstone files:

```
short = mv.Network ('short.s1p')
delay_short = mv.Network ('delay_short.s1p')
```

2.2 Important Properties

The important qualities of a network are the which is referenced by the properties:

- **s**: Scattering Parameter matrix.
- **frequency**: Frequency Object.
- **z0**: Characterisic Impedance matrix.

2.3 Element-wise Operations

Simple element-wise mathematical operations on the scattering parameter matrices are accesable through overloaded operators:

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

These have various uses. For example, the difference operation returns a network that represents the complex distance between two networks. This can be used to calculate the euclidean norm between two networks like

```
(short- delay_short).s_mag
```

or you can plot it:

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

Another use is calculating or plotting de-trended phase using the division operator. This can be done by:

```
detrended_phase = (delay_short/short).s_deg
(delay_short/short).plot_s_deg()
```

2.4 Cascading and Embeding Operations

Cascading and de-embeding 2-port Networks is done so frequently, that it can also be done though operators. The cascade function is called by the power operator, `**`, and the de-embed function is done by cascading the inverse of a network, which is implemented by the property `inv`. Given the following Networks:

```
cable = mv.Network('cable.s2p')
dut = mv.Network('dut.s1p')
```

Perhaps we want to calculate a new network which is the cascaded connection of the two individual Networks *cable* and *dut*:

```
cable_and_dut = cable ** dut
```

or maybe we want to de-embed the *cable* from *cable_and_dut*:

```
dut = cable.inv ** cable_and_dut
```

You can check my functions for consistency using the equality operator

```
dut == cable.inv (cable ** dut)
```

if you want to de-embed from the other side you can use the flip() function provided by the Network class:

```
dut ** (cable.inv).flip()
```

2.5 Sub Networks

Frequently, the individual responses of a higher order network are of interest. Network type provide way quick access like so:

```
reflection_off_cable = cable.s11
transmission_through_cable = cable.s21
```

2.6 Connecting Multi-ports

mwavepy supports the connection of arbitrary ports of N-port networks. It does this using an algorithm call sub-network growth. This connection process takes into account port impedances. Terminating one port of a ideal 3-way splitter can be done like so:

```
tee = mv.Network('tee.s3p')
delay_short = mv.Network('delay_short.s1p')
```

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

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

CHAPTER

THREE

SLOW INTRODUCTION

This is a slow intro to get readers who arent especially familiar with python comfortable working with **mwavepy**. If you are familiar with python, or are impatient see the *Quick Introduction*.

mwavepy, like all of python, can be used in scripts or through the python interpreter. If you are new to python and don't understand anything on this page, please see the Install page first. From a python shell or similar (ie IPython), the **mwavepy** module can be imported like so:

```
import mwavepy as mv
```

From here all **mwavepy**'s functions can be accessed through the variable 'mv'. Help can be accessed through pythons help command. For example, to get help with the Network class

```
help(mv.Network)
```

The Network class is a representation of a n-port network. The most common way to initialize a Network is by loading data saved in a touchstone file. Touchstone files have the extension '.sNp', where N is the number of ports of the network. To create a Network from the touchstone file 'horn.s1p':

```
horn = mv.Network('horn.s1p')
```

From here you can tab out the contents of the newly created Network by typing `horn.[hit tab]`. You can get help on the various functions as described above. The base storage format for a Network's data is in scattering parameters, these can be accessed by the property, 's'. Basic element-wise arithmetic can also be done on the scattering parameters, through operations on the Networks themselves. For instance if you want to form the complex division of two Networks scatering matrices,

This can also be used to implement averaging

Other non-elementwise operations are also available, such as cascading and de-embeding two-port networks. For instance the composit network of two, two-port networks is formed using the power operator (`**`),

De-embeding can be accomplished by using the floor division (`//`) operator

CHAPTER

FOUR

CALIBRATION

4.1 Intro

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

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

Calibrations are performed through a Calibration class, which makes creating and working with calibrations easy. Since mwavepy-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 mwavepy will determine it through inspection.

4.2 One-Port

See *One-Port 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 mwavepy as mv


## created necessary data for Calibration class

# a list of Network types, holding 'ideal' responses
```

```
my_ideals = [\
        mv.Network('ideal/short.s1p'),
        mv.Network('ideal/open.s1p'),
        mv.Network('ideal/load.s1p'),
        ]

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

## create a Calibration instance
cal = mv.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 = mv.Network('my_dut.s1p')
dut_caled = cal.apply_cal(dut)

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

Concise One-port

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

```
import mwavepy as mv

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


## create a Calibration instance
cal = mv.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
```

4.3 Two-port

Two-port calibration is more involved than one-port. mwavepy supports two-port calibration using a 8-term error model based on the algorithm described in “A Generalization of the TSD Network-Analyzer Calibration Procedure, Covering n-Port Scattering-Parameter Measurements, Affected by Leakage Errors" by R.A. Speciale here.

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). A note on switch-terms

Switch-terms are explained in Roger Marks's paper titled 'Formulations of the Basic Vector Network Analyzer Error Model including Switch-Terms' here. 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. I have support for switch terms in my HP8510C class here, which you can use or extend to different VNA. Without switch-term measurements, your calibration quality will vary depending on properties of you VNA.

See *Two-Port Calibration* for examples

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 (S21=S12=0). So if you measure reflective standards you must measure two of them simultaneously, and store information in a two-port. For example, connect a short to port-1 and a load to port-2, and save a two-port measurement as 'short,load.s2p' or similar:

```
import mwavepy as mv


## created necessary data for Calibration class

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

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


## create a Calibration instance
cal = mv.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 = mv.Network('my_dut.s2p')
dut_caled = cal.apply_cal(dut)

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

4.5 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 mwavepy'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 = mv.Network('ideals/short.s1p')
load = mv.Network('ideals/load.s1p')
short_load = wb.two_port_reflect(short, load)
```

CHAPTER

FIVE

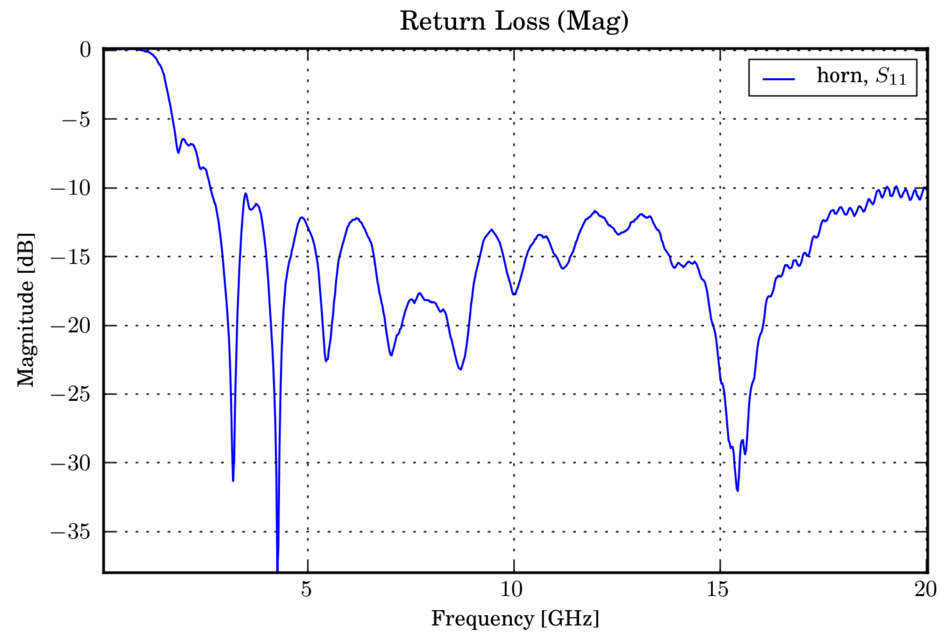
EXAMPLES

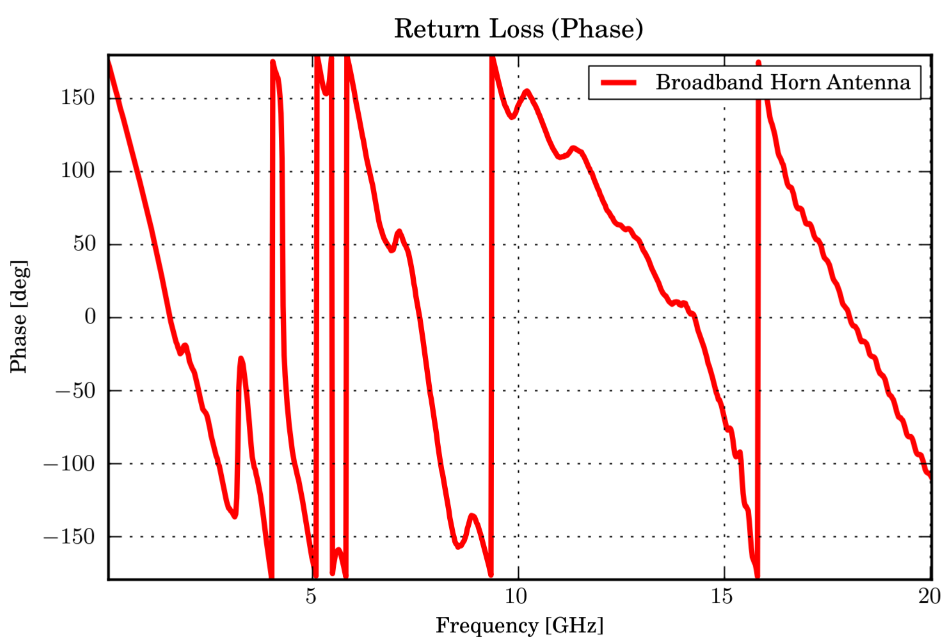
Contents:

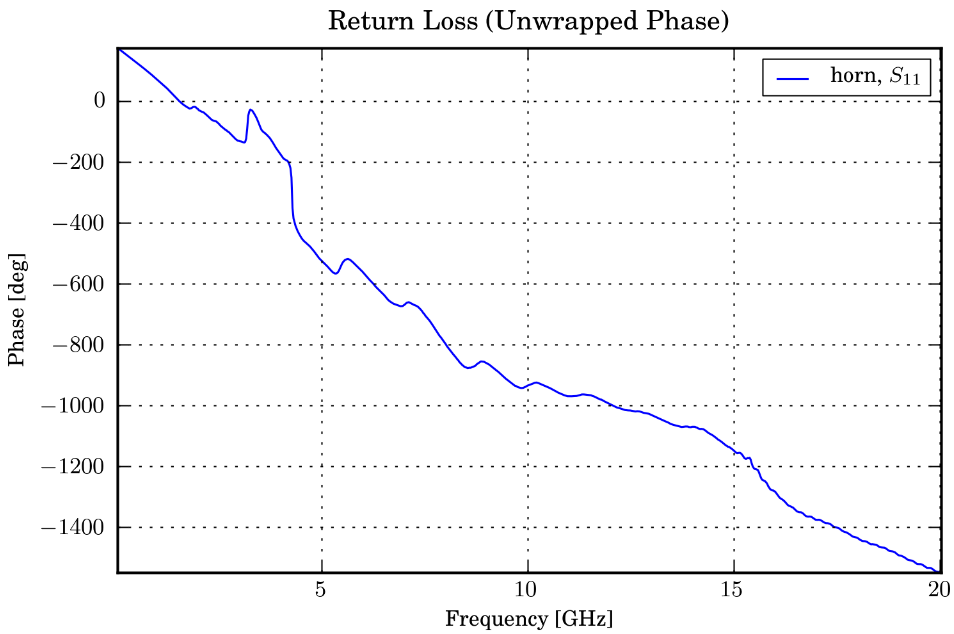
5.1 Basic Plotting

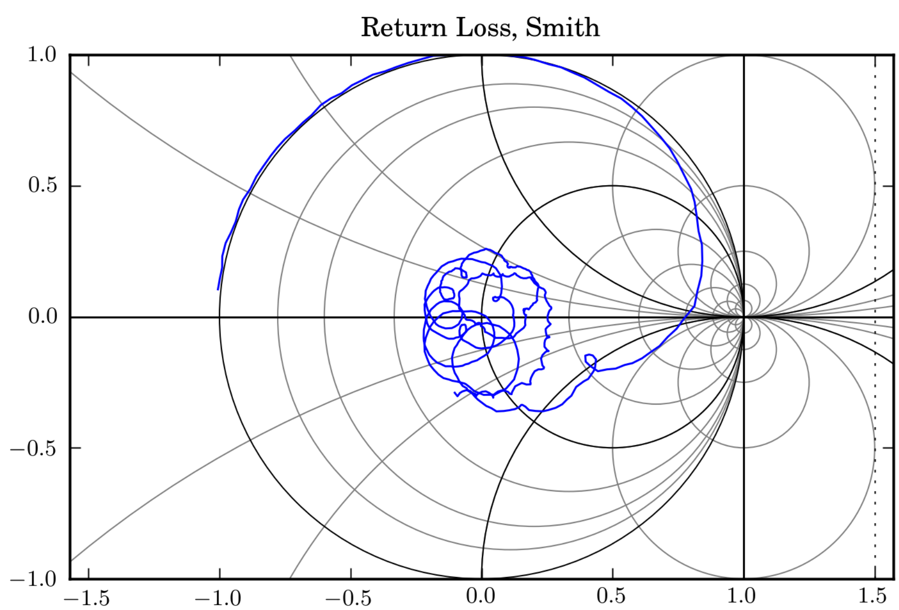
This example illustrates how to create common plots:

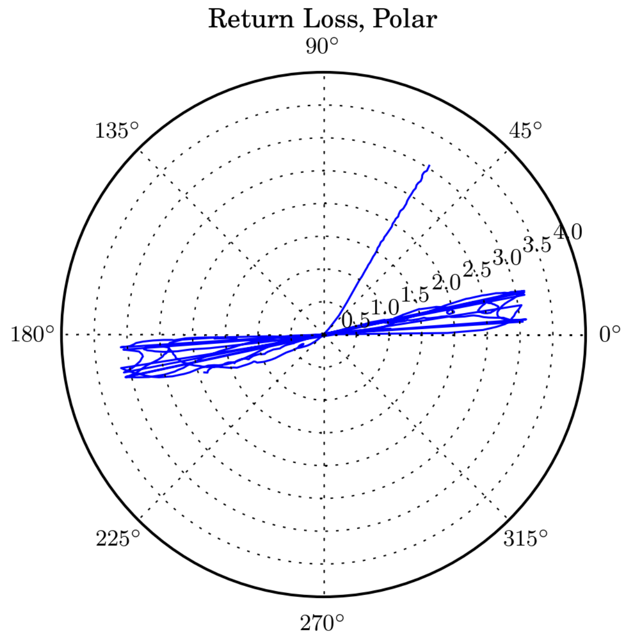
```
import mwavepy as mv
import pylab


# create a Network type from a touchstone file of a horn antenna
horn = mv.Network('horn.s1p')

# plot magnitude of S11
pylab.figure(1)
pylab.title('Return Loss (Mag)')
horn.plot_s_db(m=0,n=0) # m,n are S-Matrix indecies
# show the plots (only needed if you dont have interactive set on ipython)
pylab.show()

# plot phase of S11
pylab.figure(2)
pylab.title('Return Loss (Phase)')
# all keyword arguments are passed to matplotlib.plot command
horn.plot_s_deg(0,0, label='Broadband Horn Antenna', color='r', linewidth=2)

# plot unwrapped phase of S11
pylab.figure(3)
pylab.title('Return Loss (Unwrapped Phase)')
horn.plot_s_deg_unwrapped(0,0)

# plot complex S11 on smith chart
pylab.figure(5)
horn.plot_s_smith(0,0, show_legend=False)
pylab.title('Return Loss, Smith')

# plot complex S11 on polar grid
pylab.figure(4)
horn.plot_s_polar(0,0, show_legend=False)
pylab.title('Return Loss, Polar')
```


Return Loss (Mag)
horn, S_{11}
Magnitude [dB]
Frequency [GHz]
0
−5
−10
−15
−20
−25
−30
−35
5
10
15
20


Return Loss (Phase)
Broadband Horn Antenna
Phase [deg]
Frequency [GHz]
150
100
50
0
−50
−100
−150
5
10
15
20


Return Loss (Unwrapped Phase)
horn, S_{11}
Phase [deg]
0
−200
−400
−600
−800
−1000
−1200
−1400
5
10
15
20
Frequency [GHz]


Return Loss, Smith
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5




```
#  to save all figures,
mv.save_all_figs('.', format = ['png','eps'])
```

5.2 One-Port Calibration

5.2.1 Instructive

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

```
import mwavepy as mv


## created necessary data for Calibration class

# a list of Network types, holding 'ideal' responses
my_ideals = [\
        mv.Network('ideal/short.s1p'),
        mv.Network('ideal/open.s1p'),
        mv.Network('ideal/load.s1p'),
        ]

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

## create a Calibration instance
cal = mv.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 = mv.Network('my_dut.s1p')
dut_caled = cal.apply_cal(dut)

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

5.2.2 Concise

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

```
import mwavepy as mv

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


## create a Calibration instance
cal = mv.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
```

5.3 Two-Port Calibration

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

```
import mwavepy as mv


## created necessary data for Calibration class

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

# a list of Network types, holding 'measured' responses
my_measured = [\
        mv.Network('measured/thru.s2p'),
```

```
        mv.Network('measured/line.s2p'),
        mv.Network('measured/short, short.s2p'),
        ]


## create a Calibration instance
cal = mv.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 = mv.Network('my_dut.s2p')
dut_caled = cal.apply_cal(dut)

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

CHAPTER

SIX

ARCHITECTURE

6.1 Module Layout and Inheritance

6.2 Individual Class Architectures

6.2.1 Frequency

The frequency object was created to make storing and manipulating frequency information easier and more rigid. A major convenience this class provides is the acounting of the frequency vector's unit. Other objects, such as Network, and Calibration require a frequency vector to be meaningful. This vector is commonly referenced when a plot is generated, which one generally doesnt was in units of Hz. If the Frequency object did not exist other objects which require frequency information would have to implement the unit and multiplier bagage.

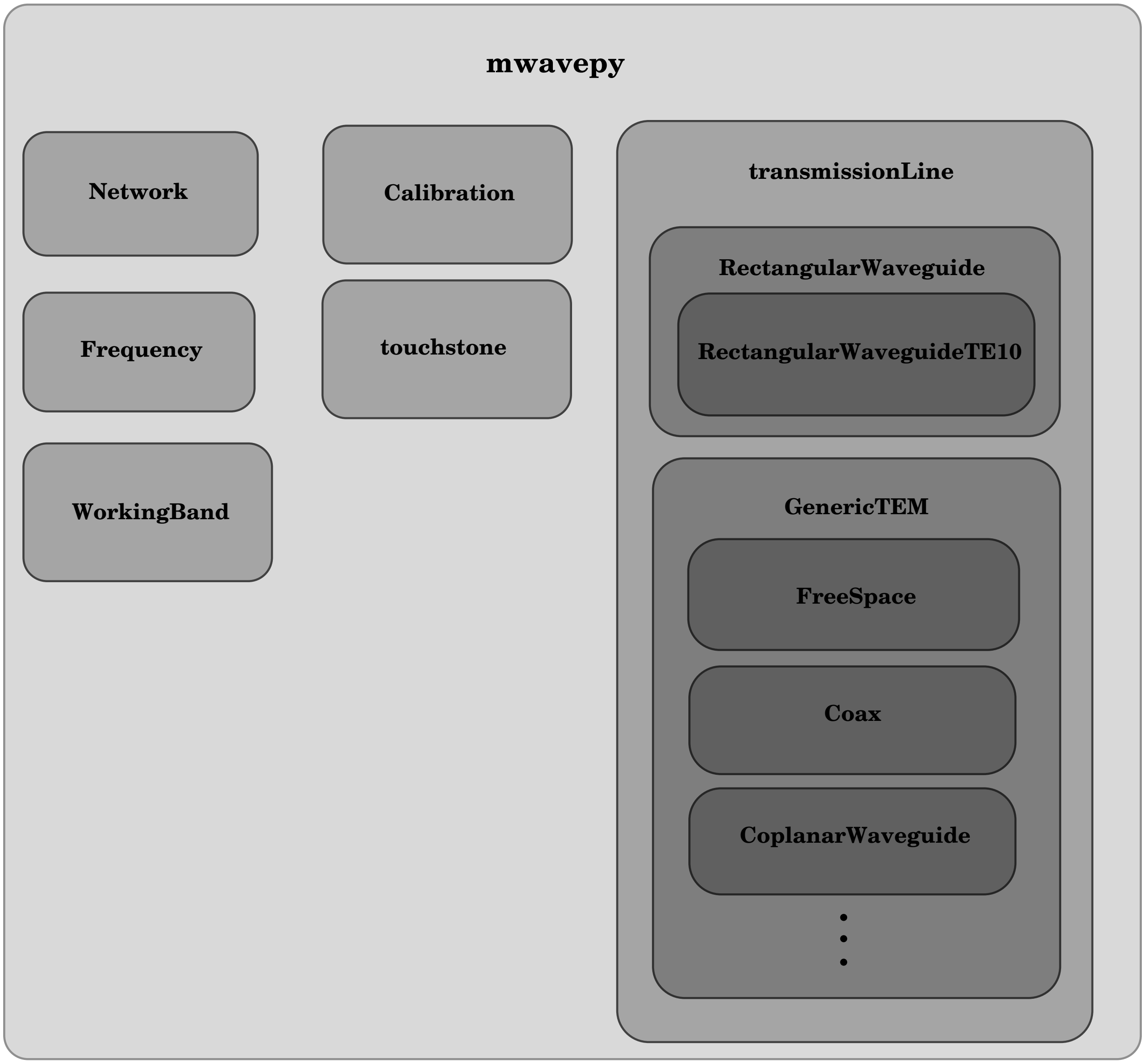
6.2.2 Network

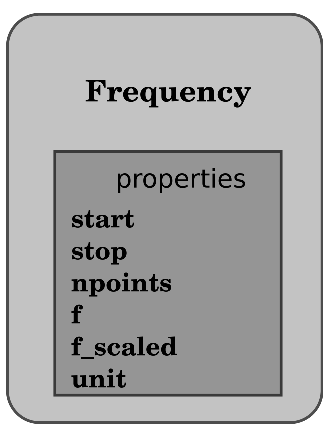
6.2.3 touchstone

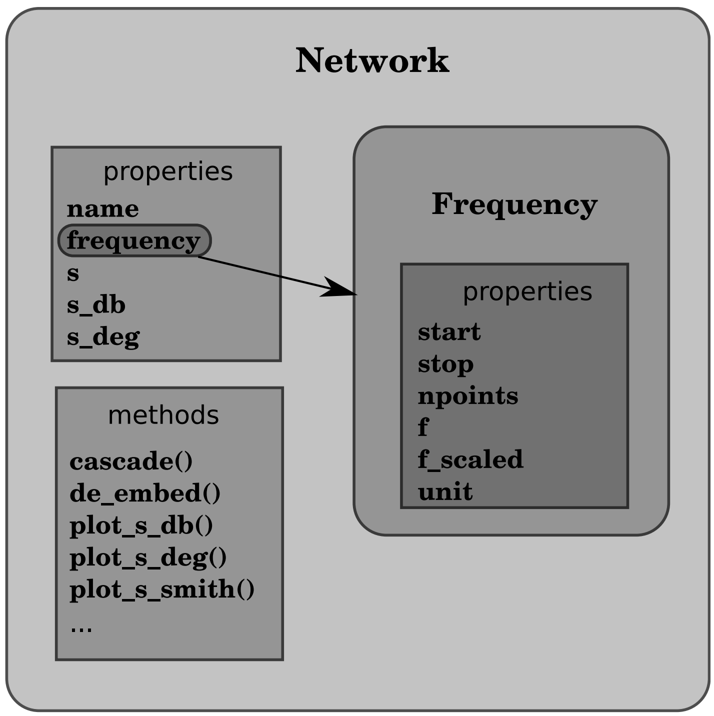
The standard file format used to store data retrieved from Vector Network Analyzers (VNAs) is the touchstone file format. This file contains all relevent data of a measured network such as frequency info, network parameters (s, y,z, etc), and port impedance.

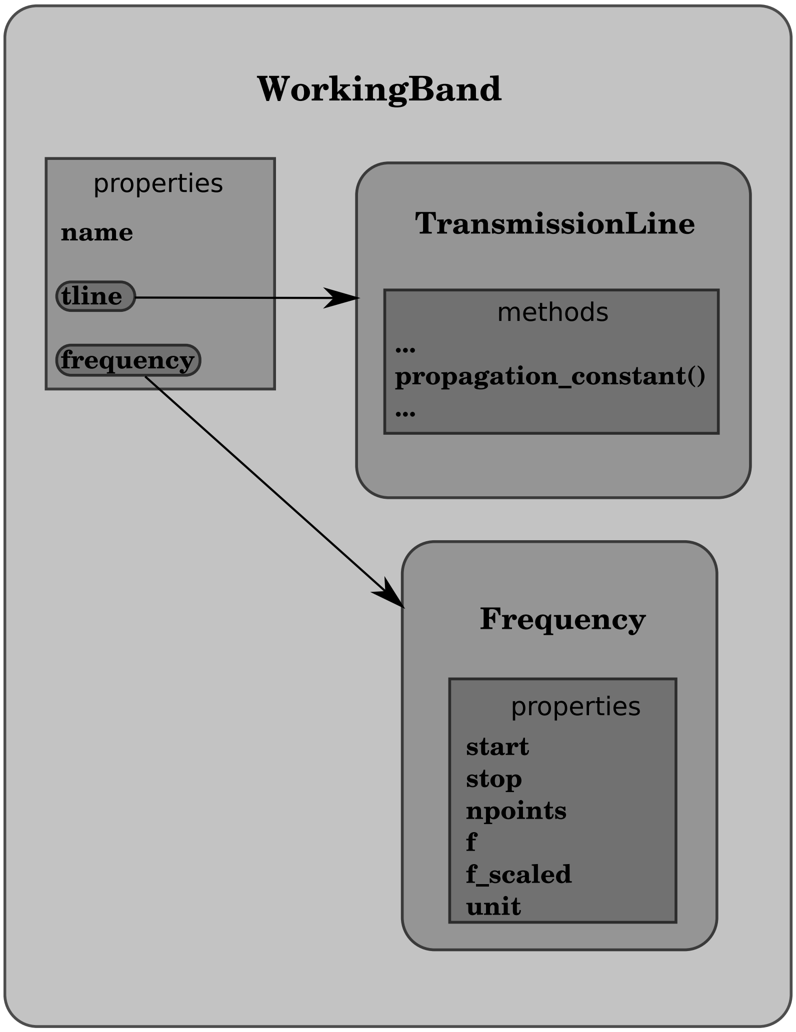
6.2.4 WorkingBand

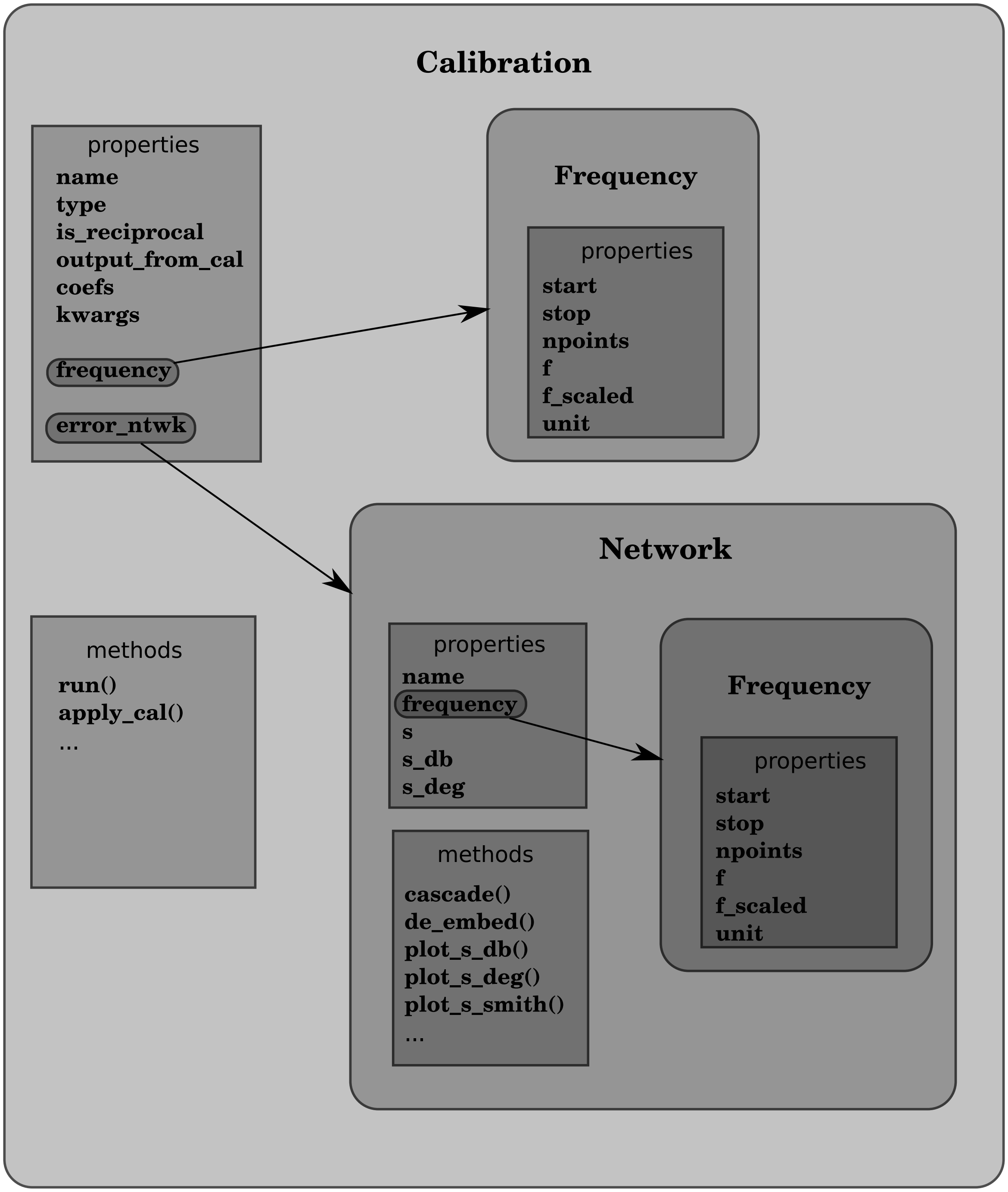
6.2.5 Calibration


mwavepy
Network
Calibration
transmissionLine
RectangularWaveguide
RectangularWaveguideTE10
Frequency
touchstone
WorkingBand
GenericTEM
FreeSpace
Coax
CoplanarWaveguide


Frequency
properties
start
stop
npoints
f
f_scaled
unit


Network
properties
name
frequency
s
s_db
s_deg
methods
cascade()
de_embed()
plot_s_db()
plot_s_deg()
plot_s_smith()
...
Frequency
properties
start
stop
npoints
f
f_scaled
unit


WorkingBand
properties
name
tline
frequency
TransmissionLine
methods
...
propagation_constant()
...
Frequency
properties
start
stop
npoints
f
f_scaled
unit


Calibration
properties
name
type
is_reciprocal
output_from_cal
coefs
kwargs
frequency
error_ntwk
Frequency
properties
start
stop
npoints
f
f_scaled
unit
methods
run()
apply_cal()
...
Network
properties
name
frequency
s
s_db
s_deg
methods
cascade()
de_embed()
plot_s_db()
plot_s_deg()
plot_s_smith()
...
Frequency
properties
start
stop
npoints
f
f_scaled
unit

CHAPTER

SEVEN

API

Major classes

7.1 Frequency

class `mwavepy.`**`Frequency`**(*start*, *stop*, *npoints*, *unit='hz'*)

represents a frequency band.

attributes: start: starting frequency (in Hz) stop: stoping frequency (in Hz) npoints: number of points, an int

unit: unit which to scale a formated axis, when accesssed. see

formattedAxis

frequently many calcluations are made in a given band , so this class is used in other classes so user doesnt have to continually supply frequency info.

`center`

`f`

returns a frequency vector in Hz

`f_scaled`

returns a frequency vector in units of self.unit

classmethod **`from_f`**(*f*)

alternative constructor from a frequency vector, in Hz takes:

f: frequency array in Hz

`labelXAxis`(*ax=None*)

`multiplier`

multiplier for formating axis

`unit`

The unit to format the frequency axis in. see formatedAxis

7.2 touchstone

`mwavepy.`**`touchstone`**

alias of `mwavepy.touchstone`

7.3 Network

class `mwavepy.Network`(*touchstone_file=None*, *name=None*)

Represents a n-port microwave network.

the most fundemental properties are:

s: scattering matrix. a kxnxn complex matrix where 'n' is number of ports of network.

z0: characteristic impedance f: frequency vector in Hz. see also frequency, which is a

Frequency object (see help on this class for more info)

The following operators are defined as follows: '+' : element-wise addition of the s-matrix '-' : element-wise subtraction of the s-matrix '*' : element-wise multiplication of the s-matrix '/' : element-wise division of the s-matrix '**': cascading of 2-port networks '//': de-embdeding of one network from the other.

various other network properties are accesable as well as plotting routines are also defined for convenience,

most properties are derived from the specifications given for touchstone files.

`add_noise_polar`(*mag_dev*, *phase_dev*, ***kwargs*)

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

takes: mag_mag: standard deviation of magnitude phase_dev: standard deviation of phase [in degrees] n_ports: number of ports. defualt to 1

returns: nothing

`change_frequency`(*new_frequency*, ***kwargs*)

`f`

the frequency vector for the network, in Hz.

`flip`()

swaps the ports of a two port

`frequency`

returns a Frequency object, see frequency.py

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

calculates an interpolated network. defualt interpolation type is linear. see notes about other interpolation types

takes: new_frequency: **kwargs: passed to scipy.interpolate.interp1d initializer.

returns: result: an interpolated Network

note:

usefule 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.

`inv`

a network representing inverse s-parameters, for de-embeding

`multiply_noise`(*mag_dev*, *phase_dev*, ***kwargs*)

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

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

returns: nothing

nudge (*amount=1e-12*)
perturb s-parameters by small amount. this is usefule to work-around numerical bugs. takes:

amount: amount to add to s parameters

returns: na

number_of_ports
the number of ports the network has.

passivity

passivity metric for a multi-port network. It returns

a matrix who's diagonals are equal to the total power received at all ports, normalized to the power at a single excitement port.

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

for two port this is (|S11|^2 + |S21|^2, |S22|^2+|S12|^2)

in general it is S.H * S

where H is conjugate transpose of S, and * is dot product

note: see more at, http://en.wikipedia.org/wiki/Scattering_parameters#Lossless_networks

plot_polar_generic (*attribute_r*, *attribute_theta*, *m=0*, *n=0*, *ax=None*, *show_legend=True*, ***kwargs*)
generic plotting function for plotting a Network's attribute in polar form

takes:

plot_s_all_db (*ax=None*, *show_legend=True*, ***kwargs*)
plots all s parameters in log magnitude

takes:

ax - matplotlib.axes object to plot on, used in case you want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_s_db (*m=None*, *n=None*, *ax=None*, *show_legend=True*, ***kwargs*)
plots the magnitude of the scattering parameter of indecies m, n in log magnitude

takes: m - first index, int n - second indext, int ax - matplotlib.axes object to plot on, used in case you

want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_s_deg (*m=None*, *n=None*, *ax=None*, *show_legend=True*, ***kwargs*)
plots the phase of a scattering parameter of indecies m, n in degrees

takes: m - first index, int n - second indext, int ax - matplotlib.axes object to plot on, used in case you

want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_s_deg_unwrapped (*m=None*, *n=None*, *ax=None*, *show_legend=True*, ***kwargs*)
plots the phase of a scattering parameter of indecies m, n in unwrapped degrees

takes: m - first index, int n - second indext, int ax - matplotlib.axes object to plot on, used in case you

want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_s_mag (*m=None*, *n=None*, *ax=None*, *show_legend=True*, ***kwargs*)
plots the magnitude of a scattering parameter of indecies m, n not in magnitude

takes: m - first index, int n - second indext, int ax - matplotlib.axes object to plot on, used in case you

want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_s_polar (*m=0*, *n=0*, *ax=None*, *show_legend=True*, ***kwargs*)
plots the scattering parameter of indecies m, n in polar form

takes: m - first index, int n - second indext, int ax - matplotlib.axes object to plot on, used in case you

want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_s_rad (*m=None*, *n=None*, *ax=None*, *show_legend=True*, ***kwargs*)
plots the phase of a scattering parameter of indecies m, n in radians

takes: m - first index, int n - second indext, int ax - matplotlib.axes object to plot on, used in case you

want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_s_rad_unwrapped (*m=None*, *n=None*, *ax=None*, *show_legend=True*, ***kwargs*)
plots the phase of a scattering parameter of indecies m, n in unwrapped radians

takes: m - first index, int n - second indext, int ax - matplotlib.axes object to plot on, used in case you

want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_s_smith (*m=None*, *n=None*, *r=1*, *ax=None*, *show_legend=True*, ***kwargs*)
plots the scattering parameter of indecies m, n on smith chart

takes: m - first index, int n - second indext, int r - radius of smith chart ax - matplotlib.axes object to plot on, used in case you

want to update an existing plot.

show_legend: boolean, to turn legend show legend of not **kwargs - passed to the matplotlib.plot command

plot_vs_frequency_generic (*attribute*, *y_label=None*, *m=None*, *n=None*, *ax=None*, *show_legend=True*, ***kwargs*)
generic plotting function for plotting a Network's attribute vs frequency.

takes:

read_touchstone (*filename*)
loads values from a touchstone file.

takes: filename - touchstone file name, string.

note: ONLY 'S' FORMAT SUPORTED AT THE MOMENT all work is tone in the touchstone class.

`s`

The scattering parameter matrix.

s-matrix has shape fxnxn, where;

f is frequency axis and, n's are port indicies

`s11`

`s12`

`s21`

`s22`

`s_db`

returns the magnitude of the s-parameters, in dB

note:

dB is calculated by 20*log10(|s|)

`s_deg`

returns the phase of the s-parameters, in radians

`s_deg_unwrap`

returns the unwrapped phase of the s-paramerts, in degrees

`s_mag`

returns the magnitude of the s-parameters.

`s_rad`

returns the phase of the s-parameters, in radians.

`s_rad_unwrap`

returns the unwrapped phase of the s-parameters, in radians.

`t`

returns the t-parameters, which are also known as wave cascading matrix.

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

write a touchstone file representing this network. the only format supported at the moment is :

HZ S RI

takes:

filename: a string containing filename without extension[None]. if 'None', then will use the network's name. if this is empty, then throws an error.

dir: the directory to save the file in. [string]. Defaults to './'

note: in the future could make possible use of the touchtone class, but at the moment this would not provide any benefit as it has not **set_** functions.

`y`

`z0`

the characteristic impedance of the network.

z0 can be may be a number, or numpy.ndarray of shape n or fxn.

7.4 WorkingBand

class `mwavepy.WorkingBand`(*tline*, *frequency=None*, *z0=1*)

A WorkingBand is an high-level object which exists solely to make working with and creation of Networks within the same band, more concise and convenient.

A WorkingBand object has three properties: frequency information (Frequency object) transmission line information (transmission line object) character impedance of medium

the methods of WorkingBand saves the user the hassle of repetitously providing a tline and frequency type for every network creation.

note: frequency and tline classes are copied, so they are passed by value and not by-reference.

`delay_load`(*Gamma0*, *d*, *unit='m'*, ***kwargs*)

creates a Network for a delayed load transmission line

takes: Gamma0: reflection coefficient of load (not in dB) d: the length (see unit argument) [number] unit: string specifying 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. the kwarg 'z0' can be used to create a line of a given impedance

returns: a 1-port Network class, representing a loaded transmission line of length d

note: this just calls, self.line(d,**kwargs) ** self.load(Gamma0, **kwargs)

`delay_open`(*d*, *unit='m'*, ***kwargs*)

creates a Network for a delayed open transmission line

takes: d: the length (see unit argument) [number] unit: string specifying 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. the kwarg 'z0' can be used to create a line of a given impedance

returns: a 1-port Network class, representing a shorted transmission line of length d

note: this just calls, self.line(d,**kwargs) ** self.open(**kwargs)

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

creates a Network for a delayed short transmission line

takes: d: the length (see unit argument) [number] unit: string specifying 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. the kwarg 'z0' can be used to create a line of a given impedance

returns: a 1-port Network class, representing a shorted transmission line of length d note: this just calls, self.line(d,**kwargs) ** self.short(**kwargs)

frequency

guess_length_of_delay_short (*aNtwk*)
guess length of physical length of a Delay Short given by aNtwk

takes:

aNtwk: a mwavepy.ntwk type . (note: if this is a measurment it needs to be normalized to the reference plane)

tline: transmission line class of the medium. needed for the calculation of propagation constant

impedance_mismatch (*z1*, *z2*, ***kwargs*)
returns a two-port network for a impedance mis-match

takes: z1: complex impedance of port 1 [number, list, or 1D ndarray] z2: complex impedance of port 2 [number, list, or 1D ndarray] **kwargs: passed to mwavepy.Network constructor

returns: a 2-port network [mwavepy.Network]

line (*d*, *unit='m'*, ***kwargs*)
creates a Network for a section of matched transmission line

takes: d: the length (see unit argument) [number] unit: string specifying 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. the kwarg 'z0' can be used to create a line of a given impedance

returns: a 2-port Network class, representing a transmission line of length d

note: the only function called from the tline class is

propagation_constant(f,d), where f is frequency in Hz and d is distance in meters. so you can use any class which provides this and it will work .

example: wb = WorkingBand(...) # create a working band object wb.line(90, 'deg', z0=50)

load (*Gamma0*, *nports=1*, ***kwargs*)
creates a Network for a Load termianting a transmission line

takes: Gamma0: reflection coefficient of load (not in db) nports: number of ports. creates a short on all ports,

default is 1 [int]

****kwargs: key word arguments passed to match(), which is** called initially to create a 'blank' network

returns: a 1-port Network class, where S = Gamma0*ones(...)

match (*nports=1*, *z0=None*, ***kwargs*)
creates a Network for a perfect matched transmission line (Gamma0=0)

takes: nports: number of ports [int] **kwargs: key word arguments passed to Network Constructor

returns: a n-port Network [mwavepy.Network]

open (*nports=1*, ***kwargs*)
creates a Network for a 'open' transmission line (Gamma0=1)

takes:

nports: number of ports. creates a short on all ports, default is 1 [int]

****kwargs: key word arguments passed to match(), which is** called initially to create a 'blank' network

returns: a n-port Network [mwavepy.Network]

short (*nports=1*, ***kwargs*)
creates a Network for a short transmission line (Gamma0=-1)

takes:

nports: number of ports. creates a short on all ports, default is 1 [int]

****kwargs: key word arguments passed to match(), which is** called initially to create a 'blank' network

returns: a n-port Network [mwavepy.Network]

splitter (*nports=3*, ***kwargs*)
returns an ideal, lossless n-way splitter.

takes: nports: number of ports [int] **kwargs: key word arguments passed to match(), which is

called initially to create a 'blank' network.

returns: a n-port Network [mwavepy.Network]

tee (***kwargs*)
makes a ideal, lossless tee. (aka three port splitter)

takes:

****kwargs: key word arguments passed to match(), which is** called initially to create a 'blank' network.

returns: a 3-port Network [mwavepy.Network]

note: this just calls splitter(3)

theta_2_d (*theta*, *deg=True*)
converts electrical length to physical distance

takes: theta: electrical length, (see deg for unit)[number] deg: is theta in degrees? [boolean]

returns: d: physical distance in meters

note: this calls the function electrical_length_2_distance which

is provided by transmissionLine.functions.py

thru (***kwargs*)
creates a Network for a thru

takes:

****kwargs: key word arguments passed to match(), which is** called initially to create a 'blank' network

returns: a 2-port Network class, representing a thru

note: this just calls self.line(0)

tline

two_port_reflect (*ntwk1*, *ntwk2*, ***kwargs*)
generates a two-port reflective (S21=S12=0) network,from the responses of 2 one-port networks

takes: ntwk1: Network type, seen from port 1 ntwk2: Network type, seen from port 2

returns: result: two-port reflective Network type

example: wb.two_port_reflect(wb.short(), wb.match())

white_gaussian_polar (*phase_dev*, *mag_dev*, *n_ports=1*, ***kwargs*)
creates a complex zero-mean gaussian white-noise signal of given standard deviations for phase and magnitude

takes: phase_mag: standard deviation of magnitude phase_dev: standard deviation of phase n_ports: number of ports. defualt to 1 **kwargs: passed to Network() initializer

returns: result: Network type

7.5 Calibration

class mwavepy.**Calibration** (*measured*, *ideals*, *type=None*, *frequency=None*, *is_reciprocal=False*, *switch_terms=None*, *name=None*, ***kwargs*)
Represents a calibration instance, a class to hold sets of measurements, ideals, and calibration results.

see init for more information on usage.

note: all calibration algorithms are in calibrationAlgorithms.py, and are referenced by the dictionary in this object called 'calibration_algorihtm_dict'

Ts
T-matricies used for de-embeding.

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.

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

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

this string.

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

returns:

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

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 TBD

`error_ntwk`
a Network type which represents the error network being calibrated out.

`frequency`

`nports`
the number of ports in the calibration

`output_from_cal`
a dictionary holding all of the output from the calibration algorithm

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

`plot_residuals_db`(*ax=None*, *show_legend=True*, ***kwargs*)

plot magnitude of the resdiues, if calibration is overdetermined

`residuals`

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

`run`()
runs the calibration algorihtm.

this is automatically called the

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

`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.

'one port xds': self-calibration of a unknown-length delay-shorts.

note: algorithms referenced by calibration_algorithm_dict

CHAPTER

EIGHT

INDICES AND TABLES

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

INDEX

S

T

U

W

Y

Z