mwavepy Documentation

Release 1.4

alex arsenovic

November 21, 2011

CONTENTS

Tutorials:

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 of basic mwavepy usage. It 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
```

The most fundamental object mwavpey is a n-port *Network*. Commonly a Network is constructed from data stored in a touchstone files, like so.:

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

2.2 Important Properties

The important qualities of a *Network* are provided by the properties:

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

2.3 Element-wise Operations (Linear)

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 (Non-linear)

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 `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 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.

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).

4.3.1 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'*. 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 you VNA.

See `example_twoport_calibration` for and example

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()
```

4.4.1 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 = mv.two_port_reflect(short, load)
```

CHAPTER

FIVE

CIRCUIT DESIGN

5.1 Intro

mwavepy has basic support for microwave circuit design. Network synthesis is accomplished through the Media Class (`mwavepy.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 repitously providing frequency information for each new network created.

5.2 Media's Supported by mwavepy

Below is a list of mediums types supported by mwavepy,

- DistributedCircuit
- Freespace
- RectangularWaveguide
- CPW

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

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

```
import mwavepy as mv
freq = mv.Frequency(10,20,101,'ghz')
my_media = mv.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 = mv.Frequency(500,750,101,'ghz')
my_media = mv.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 = mv.Frequency(75,110,101,'ghz')
my_media = mv.media.RectangularWaveguide(freq, a=100*mil)
```

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 mwavepy 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.

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 mwavepy's network compnents 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 indecies (starting from 0) to be connected.

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

```
ntwkC = mv.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 conect 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)
```

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 = mv.zl_2_Gamma0(z0=50,zl=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()
```

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 somthing we want to minimize, such as the reflection coefficient magnitude at band center.

```
from scipy.optmize import fmin

# the load we are trying to match
load = my_media.load(mv.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))
```

Examples:

CHAPTER

SIX

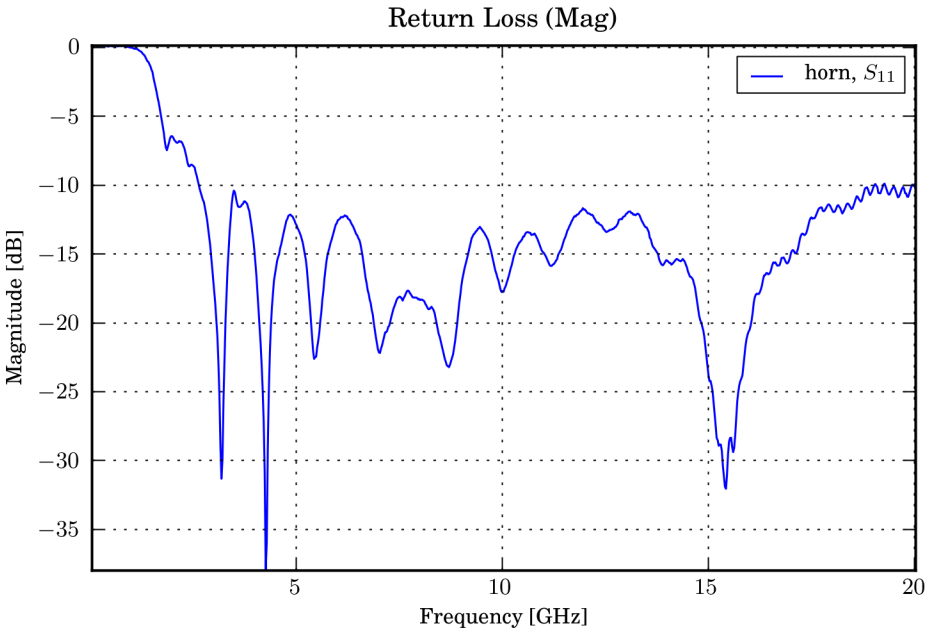
EXAMPLES

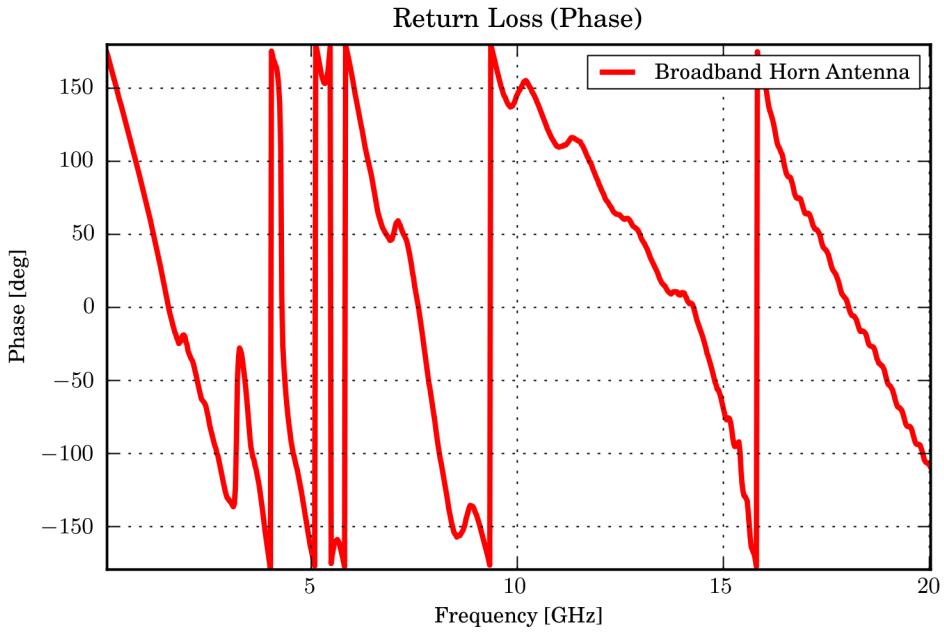
Contents:

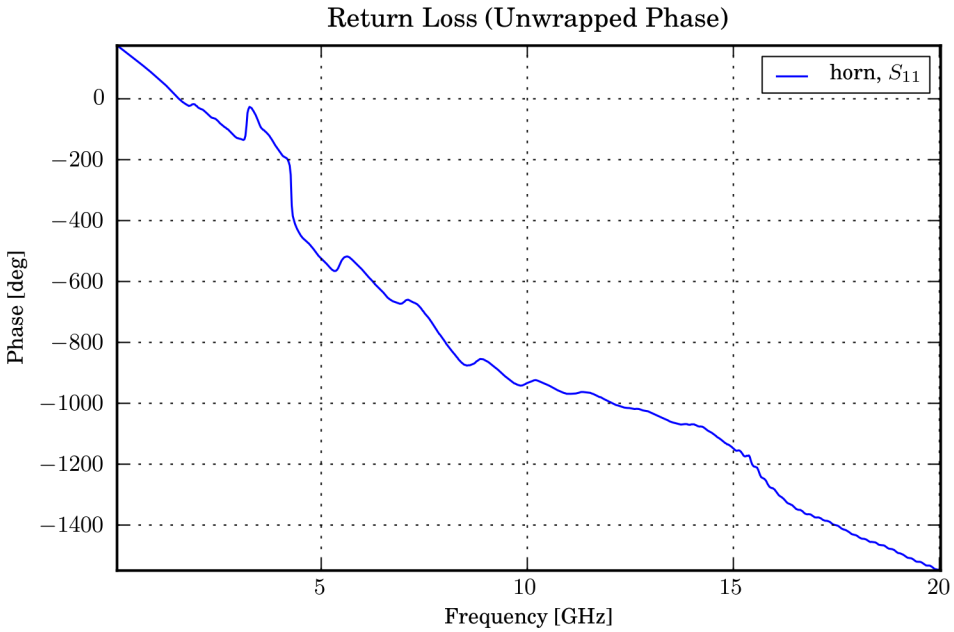
6.1 Basic Plotting

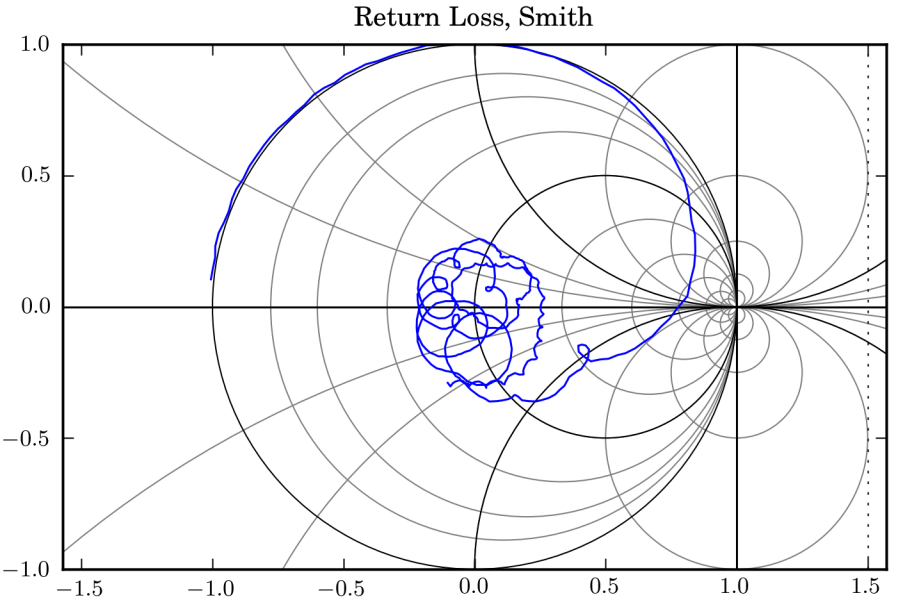
This example illustrates how to create common plots:

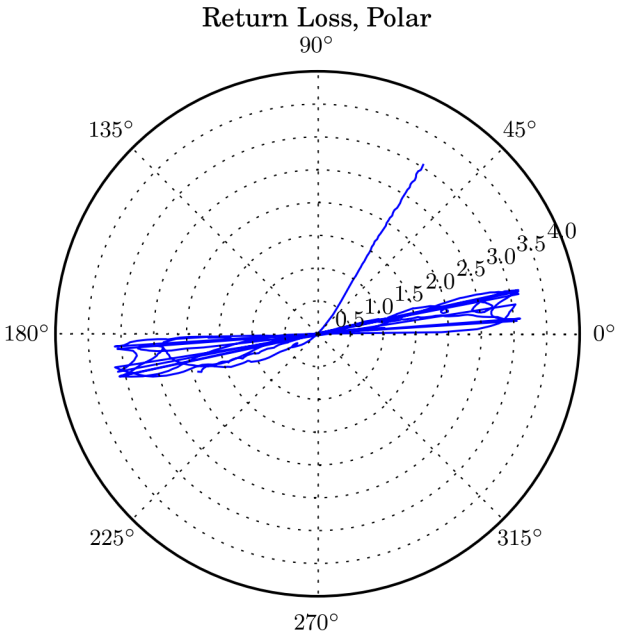
```
import mwavepy as mv
import pylab


# 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]
0
−5
−10
−15
−20
−25
−30
−35
5
10
15
20
Frequency [GHz]


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


Return Loss (Unwrapped Phase)
horn, S_{11}
0
−200
−400
−600
−800
−1000
−1200
−1400
Phase [deg]
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'])
```

6.2 One-Port Calibration

6.2.1 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()
```

6.2.2 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
```

6.3 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()
```

6.4 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

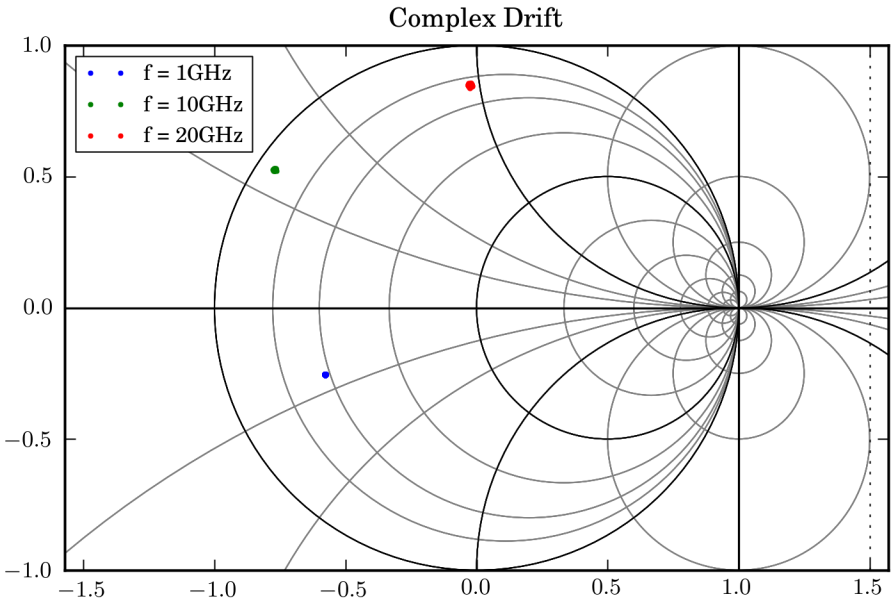
6.4.1 Touchstone File Retrieval

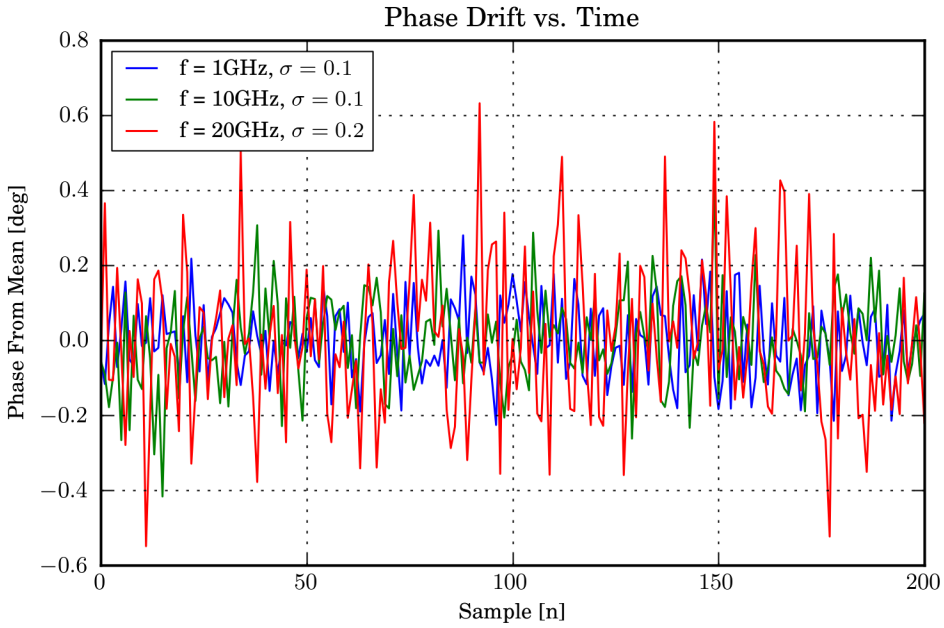
```
import mwavepy as mv
import os,datetime

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

myvna = mv.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()
```

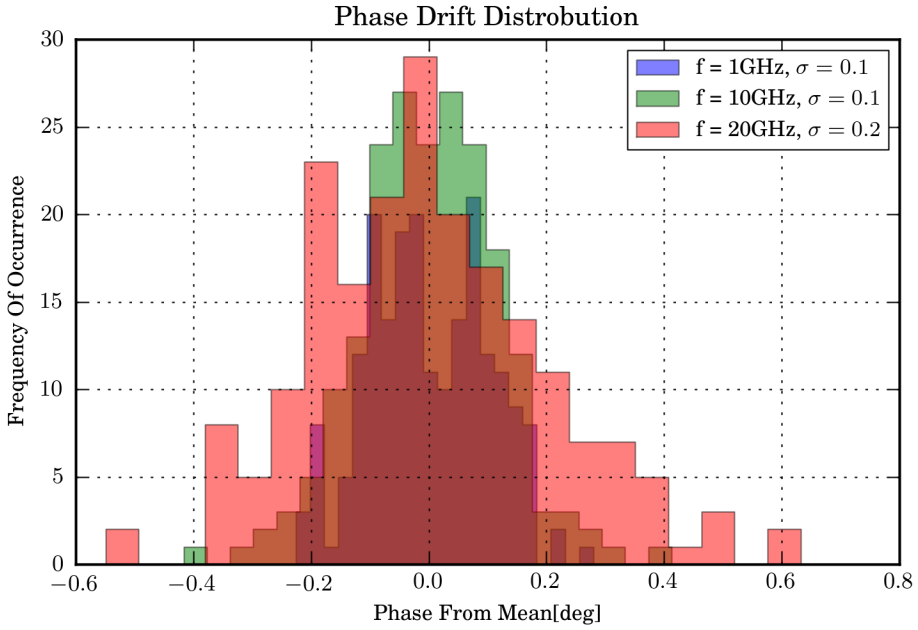
6.4.2 Noise Analysis

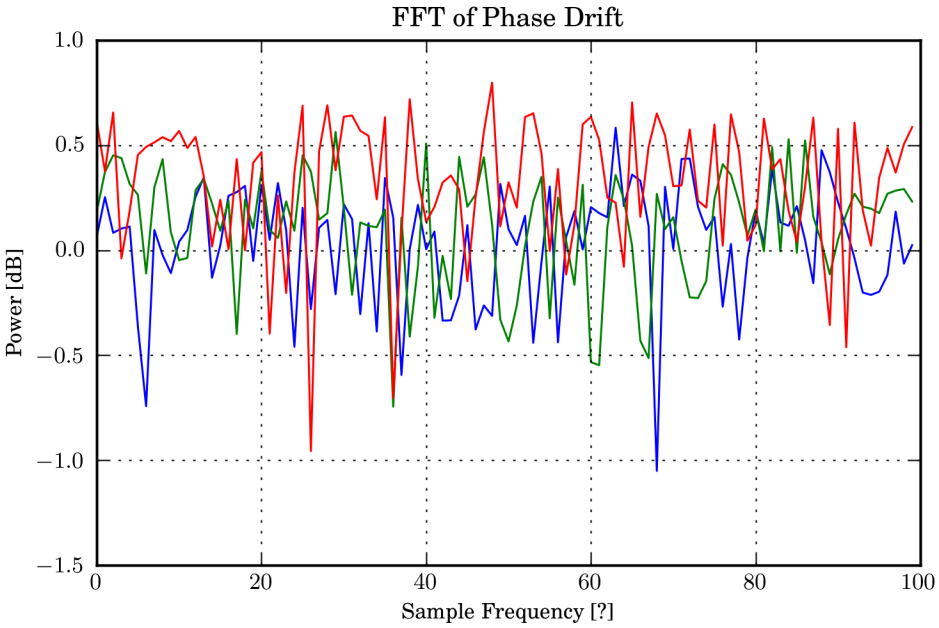
Calculates and plots various metrics of noise, given a directory of touchstones files, as would be created from the previous script

```
import mwavepy as mv
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 = mv.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 touchsone files
        z = array( [(data[keys[k]]).s[f,0,0] for k in range(len(keys))])
        phase_change = mv.complex_2_degree(z * 1/z[0])
        phase_change = phase_change - mean(phase_change)
        mag_change = mv.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()
        mv.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 Distrobution')
        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();
```


Complex Drift
f = 1GHz
f = 10GHz
f = 20GHz
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5


Phase Drift vs. Time
f = 1GHz, σ = 0.1
f = 10GHz, σ = 0.1
f = 20GHz, σ = 0.2
Phase From Mean [deg]
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0
50
100
150
200
Sample [n]

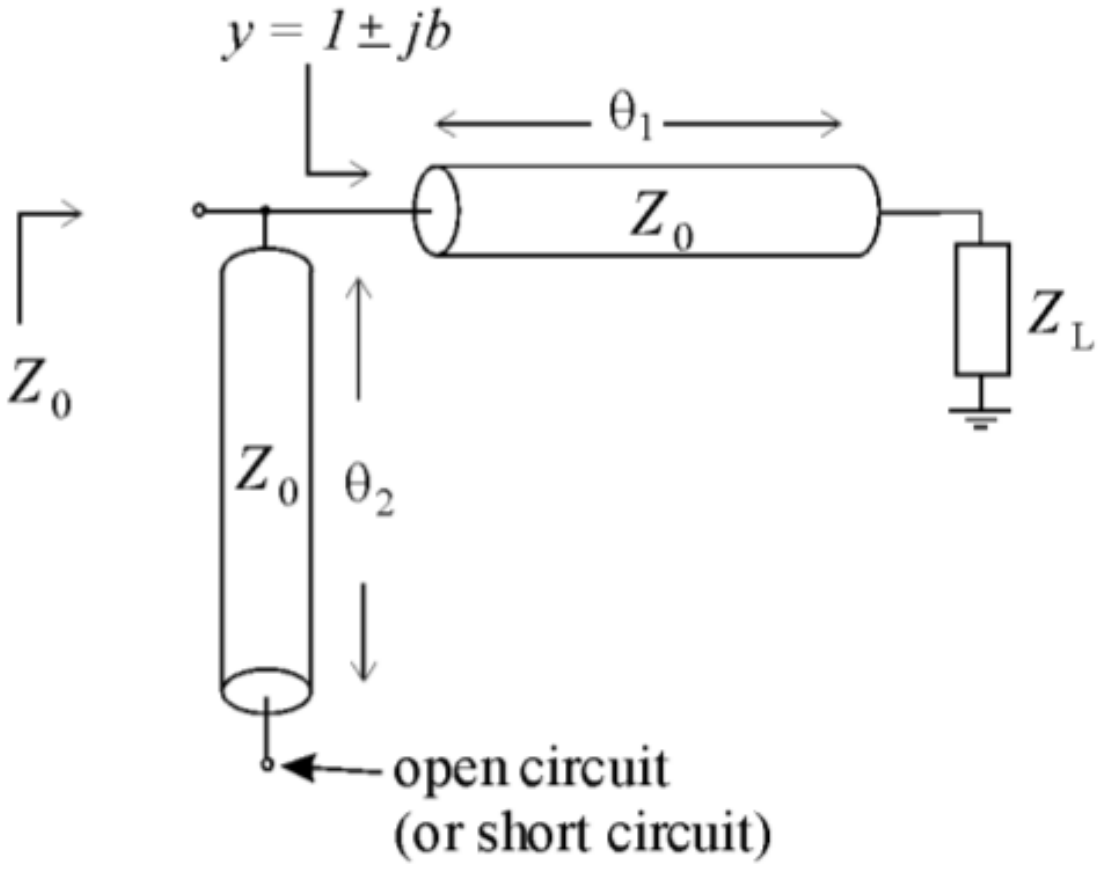

Phase Drift Distrobution
f = 1GHz, σ = 0.1
f = 10GHz, σ = 0.1
f = 20GHz, σ = 0.2
Frequency Of Occurrence
Phase From Mean[deg]
30
25
20
15
10
5
0
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
0.8


FFT of Phase Drift
Power [dB]
Sample Frequency [?]
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
20
40
60
80
100

6.5 Circuit Design: Single Stub Matching Network

6.5.1 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:

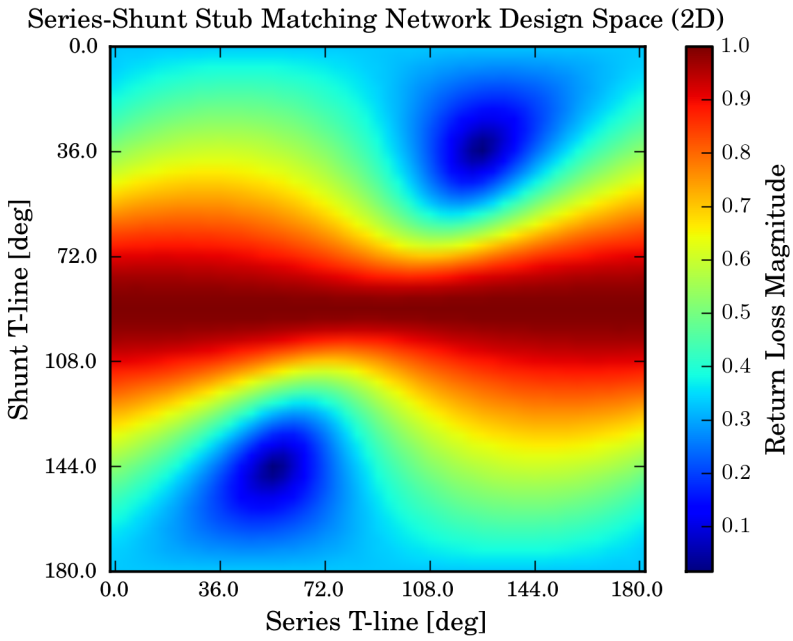
- length of series tline
- length of shunt tline

This script illustrates how to create a plot of return loss magnitude off the matched load, vs series and shunt line lengths. The optmial designs are then seen as the minima of a 2D surface.

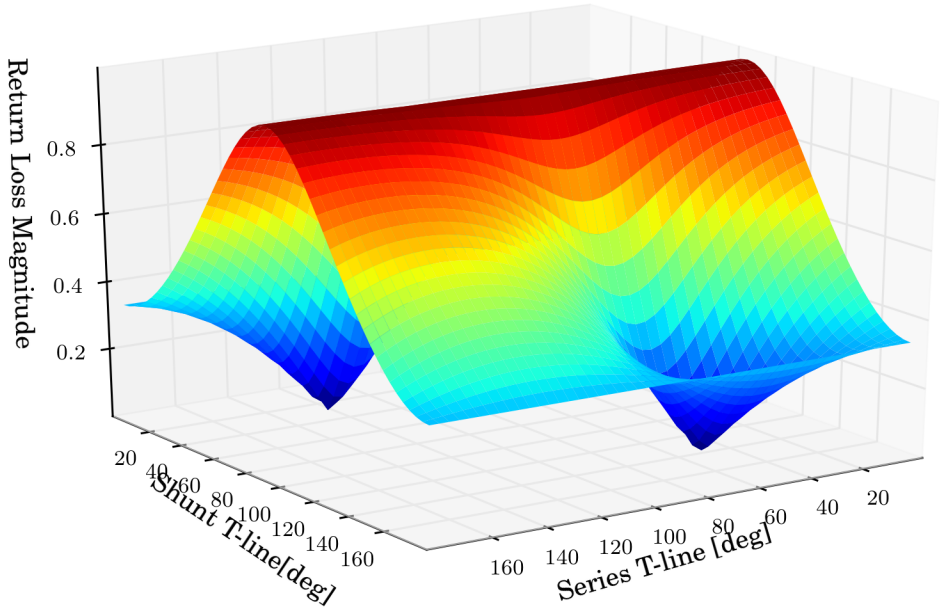
6.5.2 Script

```
import mwavepy as mv
from pylab import *


# Inputs
wg = mv.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 = mv.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(wb,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(wb,[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()
```

6.5.3 Output



Reference:


Return Loss Magnitude
0.8
0.6
0.4
0.2
20
40
60
80
100
120
140
160
Shunt T-line[deg]
160
140
120
100
80
60
40
20
Series T-line [deg]

CHAPTER

SEVEN

MWAVEPY API

7.1 mwavepy Package

7.1.1 mwavepy Package

mwavepy is an object-oriented approach to microwave engineering, implemented in the Python programming language. It provides a set of objects and features which can be used to build powerful solutions to specific problems. mwavepy's abilities are; touchstone file manipulation, calibration, VNA data acquisition, circuit design and much more.

This is the main module file for mwavepy. it simply imports classes and methods. It does this in two ways; import all into the current namespace, and import modules themselves for coherent structured referencing

7.1.2 convenience Module

Holds pre-initialized class's to provide convenience. Also provides some functions, which cant be categorized as anything better than general conviniencies.

Pre-initialized classes include: Frequency Objects for standard frequency bands Media objects for standard waveguide bands,

mwavepy.convenience.**find_nearest**(*array*, *value*)
find nearest value in array. taken from http://stackoverflow.com/questions/2566412/find-nearest-value-in-numpy-array

mwavepy.convenience.**find_nearest_index**(*array*, *value*)
find nearest value in array. taken from http://stackoverflow.com/questions/2566412/find-nearest-value-in-numpy-array

mwavepy.convenience.**legend_off**(*ax=None*)
turn off the legend for a given axes. if no axes is given then it will use current axes.

mwavepy.convenience.**now_string**()

mwavepy.convenience.**plot_complex**(*z*, **args*, ***kwargs*)
plots a complex array or list in real vs imaginary.

mwavepy.convenience.**save_all_figs**(*dir='./'*, *format=*[, *'eps'*, *'pdf'*, *'png'*])

7.1.3 frequency Module

Provides the Frequency class, and related functions

class mwavepy.frequency.**Frequency**(*start*, *stop*, *npoints*, *unit='hz'*, *sweep_type='lin'*)
Bases: object

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*, **args*, ***kwargs*)
alternative constructor from a frequency vector, takes:

f: frequency array (default in Hz)

returns: mwavepy.Frequency object

labelXAxis(*ax=None*)

multiplier
multiplier for formating axis

unit
The unit to format the frequency axis in. see formatedAxis

w
angular frequency in radians

mwavepy.frequency.**f_2_frequency**(*f*)
convienience function converts a frequency vector to a Frequency object

!depricated, use classmethod from_f instead.

7.1.4 mathFunctions Module

Provides commonly used math functions.

mwavepy.mathFunctions.**complex2MagPhase**(*complx*, *deg=False*)

mwavepy.mathFunctions.**complex2ReIm**(*complx*)

mwavepy.mathFunctions.**complex2Scalar**(*input*)

mwavepy.mathFunctions.**complex2dB**(*complx*)

mwavepy.mathFunctions.**complex_2_db**(*input*)
returns the magnitude in dB of a complex number.

returns: 20*log10(|z|)

where z is a complex number

mwavepy.mathFunctions.**complex_2_degree**(*input*)
returns the angle complex number in radians.

mwavepy.mathFunctions.**complex_2_magnitude**(*input*)
returns the magnitude of a complex number.

mwavepy.mathFunctions.**complex_2_quadrature**(*z*)
takes a complex number and returns quadrature, which is (length, arc-length from real axis)

mwavepy.mathFunctions.**complex_2_radian**(*input*)
returns the angle complex number in radians.

mwavepy.mathFunctions.**complex_components**(*z*)
break up a complex array into all possible scalar components

takes: complex ndarray return:

c_real: real part c_imag: imaginary part c_angle: angle in degrees c_mag: magnitude c_arc: arclength from real axis, angle*magnitude

mwavepy.mathFunctions.**db_2_magnitude**(*input*)
converts db to normal magnitude

returns: 10**((z)/20.)

where z is a complex number

mwavepy.mathFunctions.**db_2_np**(*x*)
converts a value in nepers to dB

mwavepy.mathFunctions.**degree_2_radian**(*deg*)

mwavepy.mathFunctions.**dirac_delta**(*x*)
the dirac function.

can take numpy arrays or numbers returns 1 or 0

mwavepy.mathFunctions.**magnitude_2_db**(*input*)
converts magnitude to db

db is given by 20*log10(|z|)

where z is a complex number

mwavepy.mathFunctions.**neuman**(*x*)
neumans number

2-dirac_delta(x)

mwavepy.mathFunctions.**np_2_db**(*x*)
converts a value in dB to neper's

mwavepy.mathFunctions.**null**(*A*, *eps=1e-15*)

calculates the null space of matrix A.

i found this on stack overflow.

mwavepy.mathFunctions.**psd2TimeDomain**(*f*, *y*, *windowType='hamming'*)
convert a one sided complex spectrum into a real time-signal. takes

f: frequency array, y: complex PSD arary windowType: windowing function, defaults to rect

returns in the form: [timeVector, signalVector]

timeVector is in inverse units of the input variable f, if spectrum is not baseband then, timeSignal is modulated by

exp(t*2*pi*f[0])

so keep in mind units, also due to this f must be increasing left to right

mwavepy.mathFunctions.**radian_2_degree**(*rad*)

mwavepy.mathFunctions.**scalar2Complex**(*input*)

7.1.5 network Module

Provides the Network class and related functions.

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

Bases: object

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

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

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 keyward 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_passivity(*port=None*, *ax=None*, *show_legend=True*, **args*, ***kwargs*)
plots the passivity of a network, possible for a specific port. see the property ‘passivity’ for more information.

takes: port: list of ports[ints] to plot passivity for [None] 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 *args,**kwargs - passed to the matplotlib.plot command

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*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_complex(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***kwargs*)
plots the scattering parameter of indecies m, n on complex plane

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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_db(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_deg(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_deg_unwrap(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_deg_unwrapped(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_im(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***kwargs*)
plots the imaginary part of a scattering parameter of indecies m, n

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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_mag(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_polar(*m=0*, *n=0*, *ax=None*, *show_legend=True*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_rad(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_rad_unwrapped(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_re(*m=None*, *n=None*, *ax=None*, *show_legend=True*, **args*, ***kwargs*)
plots the real part of a scattering parameter of indecies m, n

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 *args,**kwargs - passed to the matplotlib.plot command

plot_s_smith(*m=None*, *n=None*, *r=1*, *ax=None*, *show_legend=True*, *chart_type='z'*, **args*, ***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 chart_type: string determining countour type. options are:

'z': impedance contours (default) 'y': admittance contours

*args,**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_im

returns the imaginary part of the s-parameters.

s_mag

returns the magnitude of the s-parameters.

s_quad

this is like a arc-length, its s_quad = s_rad * s_mag

used in calculating differences in phase, but in units of distance

s_rad

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

s_rad_unwrap

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

s_re
returns the real part of the s-parameters.

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.

mwavepy.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

takes: list_of_networks: a list of Networks

returns: ntwk: the resultant averaged Network [mwavepy.Network]

mwavepy.network.**cascade**(*a*, *b*)
DEPRECATED. see connect_s() instead.

cascade two s-matricies together.

a's port 2 == b's port 1

if you want a different port configuration use the flip() fuction takes:

a: a 2x2 or kx2x2 s-matrix b: a 2x2, kx2x2, 1x1, or kx1x1 s-matrix

note: BE AWARE! this relies on s2t function which has a inf problem if s11 or s22 is 1.

mwavepy.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.

takes: ntwkA: network 'A', [mwavepy.Network] k: port index on ntwkA [int] (port indecies start from 0) ntwkB: network 'B', [mwavepy.Network] l: port index on ntwkB [int]

returns: ntwkC': new network of rank (ntwkA.nports+ntwkB.nports -2)-ports

note: see functions connect_s() and innerconnect_s() for actual

S-parameter connection algorithm.

the effect of mis-matched port impedances is handled by inserting

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

mwavepy.network.**connect_s**(*S*, *k*, *T*, *l*)
connect two n-port networks together. specifically, connect port 'k' on network 'S' to port 'l' on network 'T'. The resultant network has (S.rank + T.rank-2)-ports

takes: S: S-parameter matrix [numpy.ndarray]. k: port index on S (port indecies start from 0) [int] T: S-parameter matrix [numpy.ndarray] l: port index on T [int]

returns: S': new S-parameter matrix [numpy.ndarry]

note: shape of S-parameter matrices can be either nxn, or fxnxn, where

f is the frequency axis. internally, this function creates a larger composite network

and calls the innerconnect() function. see that function for more details about the implementation

mwavepy.network.**csv_2_touchstone**(*filename*)
converts a csv file saved from a Rhode swarz and possibly other

takes: filename: name of file

returns: Network object

mwavepy.network.**de_embed**(*a*, *b*)
de-embed a 2x2 s-matrix from another 2x2 s-matrix

c = b**-1 * a

note: BE AWARE! this relies on s2t function which has a inf problem if s11 or s22 is 1.

mwavepy.network.**flip**(*a*)
invert the ports of a networks s-matrix, 'flipping' it over

note: only works for 2-ports at the moment

mwavepy.network.**fon**(*ntwk_list*, *func*, *attribute='s'*, **args*, ***kwargs*)
Applies a function to some attribute of aa list of networks, and returns the result in the form of a Network. This means information that may not be s-parameters is stored in the s-matrix of the returned Network.

takes: ntwk_list: list of mwavepy.Network types func: function to operate on ntwk_list s-matrices attribute: attribute of Network's in ntwk_list for func to act on *args: passed to func **kwargs: passed to func

returns:

mwavepy.Network type, with s-matrix the result of func, operating on ntwk_list's s-matrices

example:

averaging can be implemented with func_on_networks by func_on_networks(ntwk_list,mean)

mwavepy.network.**func_on_networks**(*ntwk_list*, *func*, *attribute='s'*, **args*, ***kwargs*)
Applies a function to some attribute of aa list of networks, and returns the result in the form of a Network. This means information that may not be s-parameters is stored in the s-matrix of the returned Network.

takes: ntwk_list: list of mwavepy.Network types func: function to operate on ntwk_list s-matrices attribute: attribute of Network's in ntwk_list for func to act on *args: passed to func **kwargs: passed to func

returns:

mwavepy.Network type, with s-matrix the result of func, operating on ntwk_list's s-matrices

example:

averaging can be implemented with func_on_networks by func_on_networks(ntwk_list,mean)

mwavepy.network.**impedance_mismatch**(*z1*, *z2*)
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]

returns: 2-port s-matrix for the impedance mis-match

mwavepy.network.**innerconnect**(*ntwkA*, *k*, *l*)
connect two ports of a single n-port network, resulting in a (n-2)-port network. port indecies start from 0.

takes: ntwk: the network. [mwavepy.Network] k: port index [int] (port indecies start from 0) l: port index [int]

returns: ntwk': new network of with n-2 ports. [mwavepy.Network]

note: see functions connect_s() and innerconnect_s() for actual

S-parameter connection algorithm.

mwavepy.network.**innerconnect_s**(*S*, *k*, *l*)
connect two ports of a single n-port network, resulting in a (n-2)-port network.

takes: S: S-parameter matrix [numpy.ndarray] k: port index [int] l: port index [int]

returns: S': new S-parameter matrix [numpy.ndarry]

This function is based on the algorithm presented in the paper:

'Perspectives in Microwave Circuit Analysis' by R. C. Compton and D. B. Rutledge

The original algorithm is given in 'A NEW GENERAL COMPUTER ALGORITHM FOR S-MATRIX CALCULATION OF INTERCONNECTED MULTIPORTS' by Gunnar Filipsson

mwavepy.network.**inv**(*s*)
inverse s-parameters, used for de-embeding

mwavepy.network.**load_all_touchstones**(*dir='.'*, *contains=None*, *f_unit=None*)
loads all touchtone files in a given dir

takes: dir - the path to the dir, passed as a string (defalut is cwd) contains - string which filename must contain to be loaded, not

used if None.(default None)

returns:

ntwkDict - a Dictonary with keys equal to the file name (without a suffix), and values equal to the corresponding ntwk types

mwavepy.network.**one_port_2_two_port**(*ntwk*)
calculates the two-port network given a symetric, reciprocal and lossless one-port network.

takes: ntwk: a symetric, reciprocal and lossless one-port network.

returns: ntwk: the resultant two-port Network

mwavepy.network.**plot_uncertainty_bounds**(*ntwk_list*, *attribute='s_mag'*, *m=0*, *n=0*, *n_deviations=3*, *alpha=0.3*, *fill_color='b'*, *std_attribute=None*, **args*, ***kwargs*)
plots mean value with +- uncertainty bounds in an Network attribute, for a list of Networks.

takes: ntwk_list: list of Netmwork types [list] attribute: attribute of Network type to analyze [string] m: first index of attribute matrix [int] n: second index of attribute matrix [int] n_deviations: number of std deviations to plot as bounds [number] alpha: passed to matplotlib.fill_between() command. [number, 0-1] *args,**kwargs: passed to **Network.plot_**'attribute' command

returns: None

Caution:

if your list_of_networks is for a calibrated short, then the

std dev of deg_unwrap might blow up, because even though each network is unwrapped, they may fall on either side fo the pi relative to one another.

mwavepy.network.**plot_uncertainty_bounds_s_db**(*ntwk_list*, *attribute='s_mag'*, *m=0*, *n=0*, *n_deviations=3*, *alpha=0.3*, *fill_color='b'*, **args*, ***kwargs*)

plots mean value with +- uncertainty bounds in an Network's attribute for a list of Networks.

This is plotted on a log scale (db), but uncertainty is calculated in the linear domain

takes: ntwk_list: list of Netmwork types [list] attribute: attribute of Network type to analyze [string] m: first index of attribute matrix [int] n: second index of attribute matrix [int] n_deviations: number of std deviations to plot as bounds [number] alpha: passed to matplotlib.fill_between() command. [number, 0-1] *args,**kwargs: passed to **Network.plot_**'attribute' command

returns: None

Caution:

if your list_of_networks is for a calibrated short, then the

std dev of deg_unwrap might blow up, because even though each network is unwrapped, they may fall on either side fo the pi relative to one another.

mwavepy.network.**plot_uncertainty_bounds_s_deg**(**args*, ***kwargs*)

this just calls plot_uncertainty_bounds(attribute= 's_deg_unwrap',*args,**kwargs)

see plot_uncertainty_bounds for help

note; the attribute 's_deg_unwrap' is called on purpose to alleviate the phase wraping effects on std dev. if you DO want to look at 's_deg' and not 's_deg_unwrap' then use plot_uncertainty_bounds

mwavepy.network.**plot_uncertainty_bounds_s_im**(**args*, ***kwargs*)

this just calls plot_uncertainty_bounds(attribute= 's_im',*args,**kwargs)

see plot_uncertainty_bounds for help

mwavepy.network.**plot_uncertainty_bounds_s_mag**(**args*, ***kwargs*)

this just calls plot_uncertainty_bounds(attribute= 's_mag',*args,**kwargs)

see plot_uncertainty_bounds for help

mwavepy.network.**plot_uncertainty_bounds_s_re**(**args*, ***kwargs*)

this just calls plot_uncertainty_bounds(attribute= 's_re',*args,**kwargs)

see plot_uncertainty_bounds for help

mwavepy.network.**s2t**(*s*)

converts a scattering parameters to 'wave cascading parameters'

input matrix shape should be should be 2x2, or kx2x2

BUG: if s -matrix has ones for reflection, thsi will produce inf's you cant cascade a matrix like this anyway, but we should handle it better

mwavepy.network.**t2s**(*t*)

converts a 'wave cascading parameters' to scattering parameters

input matrix shape should be should be 2x2, or kx2x2

mwavepy.network.**two_port_reflect**(*ntwk1*, *ntwk2*, ***kwargs*)

generates a two-port reflective (S21=S12=0) network, from the 2 one-port networks

takes: ntwk1: Network on port 1 [mwavepy.Network] ntwk2: Network on port 2 [mwavepy.Network]

returns: result: two-port reflective network, S12=S21=0 [mwavepy.Network]

example: to use a working band to create a two-port reflective standard from two one-port standards

my_media= ... two_port_reflect(my_media.short(), my_media.match())

mwavepy.network.**write_dict_of_networks**(*ntwkDict*, *dir='.'*)
writes a dictionary of networks to a given directory

7.1.6 plotting Module

provides plotting functions, which dont belong to any class.

mwavepy.plotting.**smith**(*smithR=1*, *chart_type='z'*, *ax=None*)
plots the smith chart of a given radius takes:

smithR - radius of smith chart chart_type: string representing contour type: acceptable values are

'z': lines of constant impedance 'y': lines of constant admittance

ax - matplotlib.axes instance

7.1.7 tlineFunctions Module

transmission line theory related functions

mwavepy.tlineFunctions.**Gamma0_2_Gamma_in**(*Gamma0*, *theta*)
reflection coefficient at electrical length theta takes:

Gamma0: reflection coefficient at theta=0 theta: electrical length, (may be complex)

returns: Gamma_in

note: = Gamma0 * exp(-2j* theta)

mwavepy.tlineFunctions.**Gamma0_2_zin**(*z0*, *Gamma0*, *theta*)
calculates the input impedance at electrical length theta, given a reflection coefficient and characterisitc impedance of the medium takes:

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

returns zin: input impedance at theta

mwavepy.tlineFunctions.**Gamma0_2_zl**(*z0*, *Gamma*)
calculates the input impedance given a reflection coefficient and characterisitc impedance of the medium takes:

Gamma: reflection coefficient z0 - characteristic impedance.

mwavepy.tlineFunctions.**distance_2_electrical_length**(*gamma*, *f*, *d*, *deg=False*)
calculates the electrical length of a section of transmission line.

takes:

gamma: propagation constant function [function], (a function which takes frequency in hz)

l: length of line. in meters f: frequency at which to calculate. [array-like or float]. deg: return in degrees or not. [boolean].

returns:

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

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

converts complex distributed impedance and admittance to propagation constant and characteristic impedance.

takes: distributed_admittance: what it says [complex number or array] distributed_impedance: what it says [complex number or array]

returns: propagation_constant: what it says [complex number or array] characteristic_impedance: what it says [complex number or array]

mwavepy.tlineFunctions.**electrical_length**(*gamma*, *f*, *d*, *deg=False*)

calculates the electrical length of a section of transmission line.

takes:

gamma: propagation constant function [function], (a function which takes frequency in hz)

l: length of line. in meters f: frequency at which to calculate. [array-like or float]. deg: return in degrees or not. [boolean].

returns:

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

mwavepy.tlineFunctions.**electrical_length_2_distance**(*theta*, *gamma*, *f0*, *deg=True*)

convert electrical length to a physical distance.

takes: theta: electrical length gamma: propagation constant function [function] f0: frequency of interest [number] deg: is theta in degrees [Boolean]

returns: d: physical distance

note: the gamma function must take a single variable, that is frequency and return complex propagation constant such that the propagating part is positive imag part.

mwavepy.tlineFunctions.**input_impedance_2_reflection_coefficient**(*z0*, *zl*)

calculates the reflection coefficient for a given input impedance takes:

zl: input (load) impedance [number of array]. z0 - characteristic impedance[number of array].

note: input data is typecasted to 1D complex array

mwavepy.tlineFunctions.**input_impedance_2_reflection_coefficient_at_theta**(*z0*, *zl*, *theta*)

mwavepy.tlineFunctions.**input_impedance_at_theta**(*z0*, *zl*, *theta*)

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

takes: z0 - characteristic impedance. zl: load impedance theta: electrical length of the line, (may be complex)

mwavepy.tlineFunctions.**propagation_impedance_2_distributed_circuit**(*propagation_constant*, *characteristic_impedance*)

converts complex propagation constant and characteristic impedance to distributed impedance and admittance.

takes: propagation_constant: what it says [complex number or array] characteristic_impedance: what it says [complex number or array]

returns: distributed_admittance: what it says [complex number or array] distributed_impedance: what it says [complex number or array]

mwavepy.tlineFunctions.**reflection_coefficient_2_input_impedance**(*z0*, *Gamma*)
calculates the input impedance given a reflection coefficient and characterisitc impedance of the medium takes:

Gamma: reflection coefficient z0 - characteristic impedance.

mwavepy.tlineFunctions.**reflection_coefficient_2_input_impedance_at_theta**(*z0*, *Gamma0*, *theta*)
calculates the input impedance at electrical length theta, given a reflection coefficient and characterisitc impedance of the medium takes:

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

returns zin: input impedance at theta

mwavepy.tlineFunctions.**reflection_coefficient_at_theta**(*Gamma0*, *theta*)
reflection coefficient at electrical length theta takes:

Gamma0: reflection coefficient at theta=0 theta: electrical length, (may be complex)

returns: Gamma_in

note: = Gamma0 * exp(-2j* theta)

mwavepy.tlineFunctions.**skin_depth**(*f*, *rho*, *mu_r*)
the skin depth for a material. see www.microwaves101.com for more info.

takes: f: frequency, in Hz rho: bulk resistivity of material, in ohm*m mu_r: relative permiability of material

returns: skin depth: in m

mwavepy.tlineFunctions.**surface_resistivity**(*f*, *rho*, *mu_r*)
surface resistivity. see www.microwaves101.com for more info.

takes: f: frequency, in Hz rho: bulk resistivity of material, in ohm*m mu_r: relative permiability of material

returns: surface resistivity: ohms/square

mwavepy.tlineFunctions.**theta**(*gamma*, *f*, *d*, *deg=False*)
calculates the electrical length of a section of transmission line.

takes:

gamma: propagation constant function [function], (a function which takes frequency in hz)

l: length of line. in meters f: frequency at which to calculate. [array-like or float]. deg: return in degrees or not. [boolean].

returns:

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

mwavepy.tlineFunctions.**zl_2_Gamma0**(*z0*, *zl*)
calculates the reflection coefficient for a given input impedance takes:

zl: input (load) impedance [number of array]. z0 - characteristic impedance[number of array].

note: input data is typecasted to 1D complex array

mwavepy.tlineFunctions.**zl_2_Gamma_in**(*z0*, *zl*, *theta*)

mwavepy.tlineFunctions.**zl_2_zin**(*z0*, *zl*, *theta*)
input impedance of load impedance zl at electrical length theta, given characteristic impedance z0.

takes: z0 - characteristic impedance. zl: load impedance theta: electrical length of the line, (may be complex)

7.1.8 touchstone Module

contains touchstone class (written by Werner Hoch)

class mwavepy.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

get_format(*format='ri'*)
returns the file format string used for the given format. This is usefull to get some informations.

get_noise_data()
TODO: NIY

get_noise_names()
TODO: NIY

get_sparameter_arrays()
returns the sparameters as a tuple of arrays, where the first element is 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]

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

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.

load_file(*filename*)
Load the touchstone file into the interal data structures

7.1.9 Subpackages

media Package

media Package

Provides Media super-Class and instances of Media Class's for various transmission-line mediums.

Instances of the Media Class are objects which provide methods to create network objects. See media for more detailed information.

cpw Module

contains CPW class

class mwavepy.media.cpw.**CPW**(*frequency*, *w*, *s*, *ep_r*, *t=None*, *rho=None*, **args*, ***kwargs*)

Bases: mwavepy.media.media.Media

Coplanar waveguide class

This class was made from the the documentation from the qucs project (qucs.sourceforge.net/).

K_ratio
intermediary parameter. see qucs docs on cpw lines.

Z0()
characterisitc impedance

alpha_conductor
losses due to conductor resistivity

ep_re
intermediary parameter. see qucs docs on cpw lines.

gamma()
propagation constant

k1
intermediary parameter. see qucs docs on cpw lines.

distributedCircuit Module

A transmission line defined in terms of distributed circuit components

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

Bases: mwavepy.media.media.Media

A TEM transmission line, defined in terms of distributed impedance and admittance values. This class takes the following information,

distributed Capacitance, C distributed Inductance, I distributed Resistance, R distributed Conductance, G

from these the following quantities may be calculated, which are functions of angular frequency (w):

distributed Impedance, Z'(w) = wR + jwI distributed Admittance, Y'(w) = wG + jwC

from these we can calculate properties which define their wave behavior:

characteristic Impedance, Z0(w) = sqrt(Z(w)/Y'(w)) [ohms] propagation Constant, gamma(w) = sqrt(Z(w)*Y'(w)) [none]

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

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

Y
distributed Admittance,in ohms^-1 /m

Y'(w) = wG + jwC

Z
distributed Impedance, ohms/m.

Z'(w) = wR + jwI

Z0()
The characteristic impedance in ohms

classmethod from_Media(*my_media*, **args*, ***kwargs*)
initializer which creates DistributedCircuit from a Media instance

gamma()

possibly complex propagation constant, [rad/m] gamma = sqrt(Z'*Y')

note: the components of propagation constant are interpreted as follows:

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

freespace Module

A Plane-wave in Freespace.

class mwavepy.media.freespace.**Freespace**(*frequency*, *ep_r=1*, *mu_r=1*, **args*, ***kwargs*)
Bases: mwavepy.media.distributedCircuit.DistributedCircuit

Represents a plane-wave in a homogeneous freespace, defined by [possibly complex] values of relative permativity and relative permeability.

The field properties of space are related to a disctributed circuit transmission line model given in circuit theory by:

distributed_capacitance = real(ep_0*ep_r) distributed_resistance = imag(ep_0*ep_r) distributed_inductance = real(mu_0*mu_r) distributed_conductance = imag(mu_0*mu_r)

note: this class's inheritence is; Media->DistributedCircuit->FreeSpace

media Module

Contains Media class.

class mwavepy.media.media.**Media**(*frequency*, *propagation_constant*, *characteristic_impedance*, *z0=None*)
Bases: object

The super-class for all transmission line media.

It provides methods to produce generic network components for any transmision line medium, such as line, delay_short, etc.

Network Components specific to an instance of the Media super-class such as cpw_short, microstrip_bend, are implemented within the Media instances themselves.

capacitor(*C*, ***kwargs*)
A lumped capacitor

takes: C: capacitance, in Farads, [number]

returns: mwavepy.Network

characteristic_impedance

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)

electrical_length(*d*, *deg=False*)
calculates the electrical length for a given distance, at the center frequency.

takes: d: distance, in meters deg: is d in deg?[Boolean]

returns:

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

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]

note: if z1 and z2 are arrays or lists, they must be of same length as the self.frequency.npoints

inductor(*L*, ***kwargs*)
A lumped inductor

takes: L: inductance in Henrys [number]

returns: 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

example: my_media = mwavepy.Freespace(...) my_media.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 n-port Network class, where S = Gamma0*eye(...)

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

takes: nports: number of ports [int] z0: characterisitc impedance [number of array]. defaults is

None, in which case the Media's z0 is used. Otherwise this sets the resultant network's z0. See Network.z0 property for more info

**kwargs: key word arguments passed to Network Constructor

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

example: mymatch = wb.match(2,z0 = 50, name='Super Awesome Match')

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]

propagation_constant

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]

shunt (*ntwk*, ***kwargs*)
returns a shunted ntwk. this creates a 'tee', connects 'ntwk' to port 1, and returns the result

takes: ntwk: the network to be shunted. [mwavepy.Network] **kwargs: passed to the self.tee() function

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

shunt_capacitor (*C*, **args*, ***kwargs*)
a shunt capacitor

takes: C: capacitance in farads *args: passed to self.capacitor **kwargs:passed to self.capacitor

returns: a 2-port mwavepy.Network

shunt_delay_load (**args*, ***kwargs*)
a shunted delayed load:

takes: *args: passed to self.delay_load **kwargs:passed to self.delay_load

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

shunt_delay_open (**args*, ***kwargs*)
a shunted delayed open:

takes: *args: passed to self.delay_load **kwargs:passed to self.delay_load

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

shunt_delay_short (**args*, ***kwargs*)
a shunted delayed short:

takes: *args: passed to self.delay_load **kwargs:passed to self.delay_load

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

shunt_inductor (*L*, **args*, ***kwargs*)
a shunt inductor

takes: L: inductance in henrys *args: passed to self.inductor **kwargs:passed to self.inductor

returns: a 2-port mwavepy.Network

splitter(*nports*, ***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. The electrical length is given at center frequency of self.frequency

takes:

theta: electrical length, at band center (see deg for unit) [number]

deg: is theta in degrees? [boolean]

returns: d: physical distance in meters

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)

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

z0

rectangularWaveguide Module

Rectangular Waveguide class

class mwavepy.media.rectangularWaveguide.**RectangularWaveguide**(*frequency*, *a*, *b=None*, *mode_type='te'*, *m=1*, *n=0*, *ep_r=1*, *mu_r=1*, **args*, ***kwargs*)

Bases: mwavepy.media.media.Media

Rectangular Waveguide medium.

Can be used to represent any mode of a homogeneously filled rectangular waveguide of arbitrary cross-section, mode-type, and mode index.

Z0()
the characteristic impedance of a given mode

ep
the permativity of the filling material

k0
characteristic wave number

kc
cut-off wave number

kx
eigen value in the 'a' direction

ky
eigen-value in the 'b' direction

kz()

the propagation constant, which is: IMAGINARY for propagating modes REAL for non-propagating modes,

mu
the permeability of the filling material

calibration Package

calibration Package

Provides high-level Calibration class as well as calibration algorithms and parametric standards

calibration Module

Contains the Calibration class, and supporting functions

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

Bases: object

Represents a Calibration object, which can run a calibration algorithm, store results, and apply calibration to measurements.

see init for more information on usage.

Ts
T-matricies used for de-embeding, a two-port 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.

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.

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

note:

mathematically, this is mean_s(|mean_c(r)|)

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

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:

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

frequency
frequency object for the calibration

mean_residuals()

nports
the number of ports in the calibration

nstandards
number of ideal/measurement pairs in calibration

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

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

plot_errors(*std_names=None*, **args*, ***kwargs*)
plot calibration error metrics for an over-determined calibration.

see biased_error, unbiased_error, and total_error for more info

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

takes:

attribute: name of ploting method of Network class to call

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

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

note: the residuals are calculated by:

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

plot_residuals_db(**args*, ***kwargs*)
see plot_residuals

plot_residuals_mag(**args*, ***kwargs*)
see plot_residuals

plot_residuals_smith(**args*, ***kwargs*)
see plot_residuals

plot_uncertainty_per_standard(**args*, ***kwargs*)
see uncertainty_per_standard

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'.

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

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

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

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.

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: mwavepy.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

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

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

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

note:

mathematically, this is mean_s(std_c(r))

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

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 mwavepy.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'])

mwavepy.calibration.calibration.**error_dict_2_network**(*coefs*, *frequency=None*, *is_reciprocal=False*, ***kwargs*)

convert a dictionary holding standard error terms to a Network object.

takes:

returns:

mwavepy.calibration.calibration.**rand**()

rand(d0, d1, ..., dn)

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

d0, d1, ..., dn [int] Shape of the output.

out [ndarray, shape `(d0, d1, ..., dn)`] Random values.

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to *random*.

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

mwavepy.calibration.calibration.**two_port_error_vector_2_Ts**(*error_coefficients*)

calibrationAlgorithms Module

Contains calibrations algorithms, used in the Calibration class,

mwavepy.calibration.calibrationAlgorithms.**abc_2_coefs_dict**(*abc*)

converts an abc ndarry to a dictionarry containing the error coefficients.

takes:

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

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

a, b and c are related to the error network by a = det(e) = e01*e10 - e00*e11 b = e00 c = e11

returns:

coefsDict: dictionary containing the following 'directivity':e00 'reflection tracking':e01e10 'source match':e11

note: e00 = directivity error e10e01 = reflection tracking error e11 = source match error

mwavepy.calibration.calibrationAlgorithms.**eight_term_2_one_port_coefs**(*coefs*)

mwavepy.calibration.calibrationAlgorithms.**guess_length_of_delay_short**(*aNtwk*, *tline*)

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 short plane

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

mwavepy.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.

takes:

measured - list of measured reflection coefficients. can be lists of either a kxnxn numpy.ndarray, representing a s-matrix or list of 1-port mwavepy.ntwk types.

ideals - list of assumed reflection coefficients. can be lists of either a kxnxn numpy.ndarray, representing a s-matrix or list of 1-port mwavepy.ntwk types.

returns:

a dictionary containing the following keys 'error coeffcients': dictionary containing standard error coefficients 'residuals': a matrix of residuals from the least squared

calculation. see numpy.linalg.lstsq() for more info

note: uses numpy.linalg.lstsq() for least squares calculation

see one_port_nls for a non-linear least square implementation

mwavepy.calibration.calibrationAlgorithms.**one_port_nls**(*measured*, *ideals*)

one port non-linear least squares.

takes:

measured - list of measured reflection coefficients. can be lists of either a kxnxn numpy.ndarray, representing a s-matrix or list of 1-port mwavepy.ntwk types.

ideals - list of assumed reflection coefficients. can be lists of either a kxnxn numpy.ndarray, representing a s-matrix or list of 1-port mwavepy.ntwk types.

returns:

a dictionary containing the following keys: 'error coeffcients': dictionary containing standard error coefficients 'residuals': a matrix of residuals from the least squared

calculation. see numpy.linalg.lstsq() for more info

'cov_x': covariance matrix

note: uses scipy.optmize.leastsq for non-linear least squares calculation

mwavepy.calibration.calibrationAlgorithms.**parameterized_self_calibration**(*measured*, *ideals*, *showProgress=True*, ***kwargs*)

An iterative, general self-calibration routine which can take any mixture of parameterized standards. The correct parameter values are defined as the ones which minimize the mean residual error.

takes: measured: list of Network types holding actual measurements ideals: list of ParametricStandard types showProgress: turn printing progress on/off [boolean] **kwargs: passed to minimization algorithm (scipy.optimize.fmin)

returns: a dictionary holding: '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 parametricStandard sub-module for more info on them

mwavepy.calibration.calibrationAlgorithms.**parameterized_self_calibration_bounded**(*measured*, *ideals_ps*, *showProgress=Tru* ***kwargs*)

An iterative, general self-calibration routine which can take any mixture of parameterized standards. The correct parameter values are defined as the ones which minimize the mean residual error.

takes: measured: list of Network types holding actual measurements ideals_ps: list of ParameterizedStandard types showProgress: turn printing progress on/off [boolean] **kwargs: passed to minimization algorithm (scipy.optimize.fmin)

returns: a dictionary holding: '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 ParameterizedStandard for more info on them

mwavepy.calibration.calibrationAlgorithms.**parameterized_self_calibration_nls**(*measured*, *ideals_ps*, *showProgress=True*, ***kwargs*)

An iterative, general self-calibration routine which can take any mixture of parametric standards. The correct parameter values are defined as the ones which minimize the mean residual error.

takes: measured: list of Network types holding actual measurements ideals_ps: list of ParametricStandard types showProgress: turn printing progress on/off [boolean] **kwargs: passed to minimization algorithm (scipy.optimize.fmin)

returns: a dictionary holding: '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 ParametricStandard for more info on them

mwavepy.calibration.calibrationAlgorithms.**rand**()
rand(d0, d1, ..., dn)

Random values in a given shape.

Create an array of the given shape and propagate it with random samples from a uniform distribution over `[0, 1)`.

d0, d1, ..., dn [int] Shape of the output.

out [ndarray, shape `(d0, d1, ..., dn)`] Random values.

random

This is a convenience function. If you want an interface that takes a shape-tuple as the first argument, refer to *random*.

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

`mwavepy.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 can be taken into account by passing a tuple containing the forward and reverse switch terms as 1-port Networks

takes:

measured: ordered list of measured networks. list elements should be 2-port Network types. list order must correspond with ideals.

ideals: ordered list of ideal networks. list elements should be 2-port Network types.

switch_terms: tuple of 1-port Network types holding switch terms in this order (forward, reverse).

returns:

output: a dictionary containing the follwoing keys: 'error coefficients': 'error vector': 'residuals':

note: support for gathering switch terms on HP8510C is in mwavepy.virtualInstruments.vna.py

references

Doug Rytting " Network Analyzer Error Models and Calibration Methods" RF 8 Microwave. Measurements for Wireless Applications (ARFTG/NIST)

Short Course ...

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

`mwavepy.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

Subpackages

parametricStandard Package

parametricStandard Package Provides parametric standards used in self-calibration routines

generic Module Provides generic parametric standards which dont depend on any specific properties of a given media

The naming convention for these classes is Standard_UnknownQuantity

class mwavepy.calibration.parametricStandard.generic.**DelayLoad_UnknownLength**(*media*, *d*, *Gamma0*, ***kwargs*)

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A Delayed Termination of unknown length, but known termination

class mwavepy.calibration.parametricStandard.generic.**DelayLoad_UnknownLength_UnknownLoad**(*media*, *d*, *Gamma0*, ***kwargs*)

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A Delayed load of unknown length or reflection coefficient. Assumes the load is frequency independent

class mwavepy.calibration.parametricStandard.generic.**DelayLoad_UnknownLoad**(*media*, *d*, *Gamma0*, ***kwargs*)

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A Delayed Load of unknown Load. Assumes load is frequency independent

class mwavepy.calibration.parametricStandard.generic.**DelayOpen_UnknownLength**(*media*, *d*, ***kwargs*)

Bases: mwavepy.calibration.parametricStandard.generic.DelayLoad_UnknownLength

A delay open of unknown length

class mwavepy.calibration.parametricStandard.generic.**DelayShort_UnknownLength**(*media*, *d*, ***kwargs*)

Bases: mwavepy.calibration.parametricStandard.generic.DelayLoad_UnknownLength

A delay short of unknown length

class mwavepy.calibration.parametricStandard.generic.**Line_UnknownLength**(*media*, *d*, ***kwargs*)

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A matched delay line of unknown length

initial guess for length should be given to constructor

class mwavepy.calibration.parametricStandard.generic.**Parameterless**(*ideal_network*)

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A parameterless standard.

note: this is needed so that the calibration algorithm doesnt have to handle more than one class type for standards

class mwavepy.calibration.parametricStandard.generic.**UnknownShuntCapacitance**(*media*, *C*, *ntwk*, ***kwargs*)

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A Network with unknown connector capacitance

class mwavepy.calibration.parametricStandard.generic.**UnknownShuntCapacitanceInductance**(*media*, *C*, *L*, *ntwk*, ***kwarg*

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A Network with unknown connector inductance and capacitance

class mwavepy.calibration.parametricStandard.generic.**UnknownShuntInductance**(*media*, *L*, *ntwk*, ***kwargs*)

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A Network with unknown connector inductance

parametricStandard Module Provides Parametric Standard class, and some specific instances. The specific instances are named as follows

StandardType_UnknownQuantity

exception mwavepy.calibration.parametricStandard.parametricStandard.**ParameterBoundsError**

Bases: exceptions.Exception

class mwavepy.calibration.parametricStandard.parametricStandard.**ParametricStandard**(*function=None*, *parameters={}*, *parameter_bounds={}*, ***kwargs*)

Bases: object

INF

A parametric standard represents a calibration standard which has uncertainty in its response. This uncertainty is functionally known, and represented by a parametric function, where the uknown quantity is the adjustable parameter.

This class presents an abstract interface to a general Parametric Standard. Its main purpose is to allow the self calibration routine to be independent of calibration set.

See initializer for more details.

network

a Networks instance generated by calling self.function(), for

the current set of parameters (and kwargs)

number_of_parameters
the number of parameters this standard has

parameter_array
This property provides a 1D-array interface to the parameters dictionary. This is needed to intereface teh optimizing function because it only takes a 1D-array. Therefore, order must be preserved with accessing and updating the parameters through this array. To handle this I make it return and update in alphebetical order of the parameters dictionary keys.

parameter_bounds_array
This property provides a 1D-array interface to the parameters bounds dictionary. if key doesnt exist, then i presume the parameter has no bounds. this then returns a tuple of -INF,INF where INF is a global variable in this class.

parameter_keys
returns a list of parameter dictionary keys in alphabetical order

s
a direct access to the calulated networks' s-matrix

class mwavepy.calibration.parametricStandard.parametricStandard.**SlidingLoad_UnknownTerminatio**

Bases: mwavepy.calibration.parametricStandard.parametricStandard.ParametricStandard

A set of parametersized standards representing a set of Delayed Terminations of known length, but unknown termination

rectangularWaveguide Module

virtualInstruments Package

virtualInstruments Package

import virtual instruments

futekLoadCell Module

class mwavepy.virtualInstruments.futekLoadCell.**FutekMonitor**(*ax=None*, *window_length=-1*, ***kwargs*)

Bases: object

get_data_and_plot()

update_axis_scale()

update_data()

update_line()

class mwavepy.virtualInstruments.futekLoadCell.**Futek_USB210_pipe**(*sample_rate=2.5*, *avg_len=1*)

Bases: object

close()

data

read()

write(*data='gimme datan'*)

class mwavepy.virtualInstruments.futekLoadCell.**Futek_USB210_socket**(**args*, ***kwargs*)

Bases: mwavepy.virtualInstruments.generalSocketReader.GeneralSocketReader

generalSocketReader Module

class mwavepy.virtualInstruments.generalSocketReader.**GeneralSocketReader**(*sock=None*, *sample_rate=2.5*, *avg_len=1*, *query_string='1'*, *msg_len=1000.0*)

A general class which wraps a socket and has a simple data query function, implemented by the property data_point.

this was made as a way to interface a piece of hardware which did not support gpib. is useful for general interfacing of non-standard hardware or software.

example usage: gsr = generalSocketRead() gsr.connect('127.0.0.1',1111) gsr.data_point # implicityly calls send() then receive()

close()

connect(*host*, *port*)

data

tmp = [] for n in range(self.avg_len):

sleep(1./self.sample_rate) self.send(self.query_string) tmp.append(float(self.receive()))

return npy.mean(tmp)

receive()

send(*data*)

lifetimeProbeTester Module

lifetimeProbeTesterFunctions Module

stages Module

vna Module

CHAPTER

EIGHT

INDICES AND TABLES

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

PYTHON MODULE INDEX

m

INDEX

E

F

G

I

K

L

M

N

O

S

T

U

W

Y

Z