mwavepy Documentation

Release 1.5

alex arsenovic

December 30, 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 introduction to **mwavepy** for readers who arent especially familiar with python. 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 [1], 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 a paper by Roger Marks [2]. 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'),
        ]
```

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

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

```
## 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)
```

Bibliography

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:

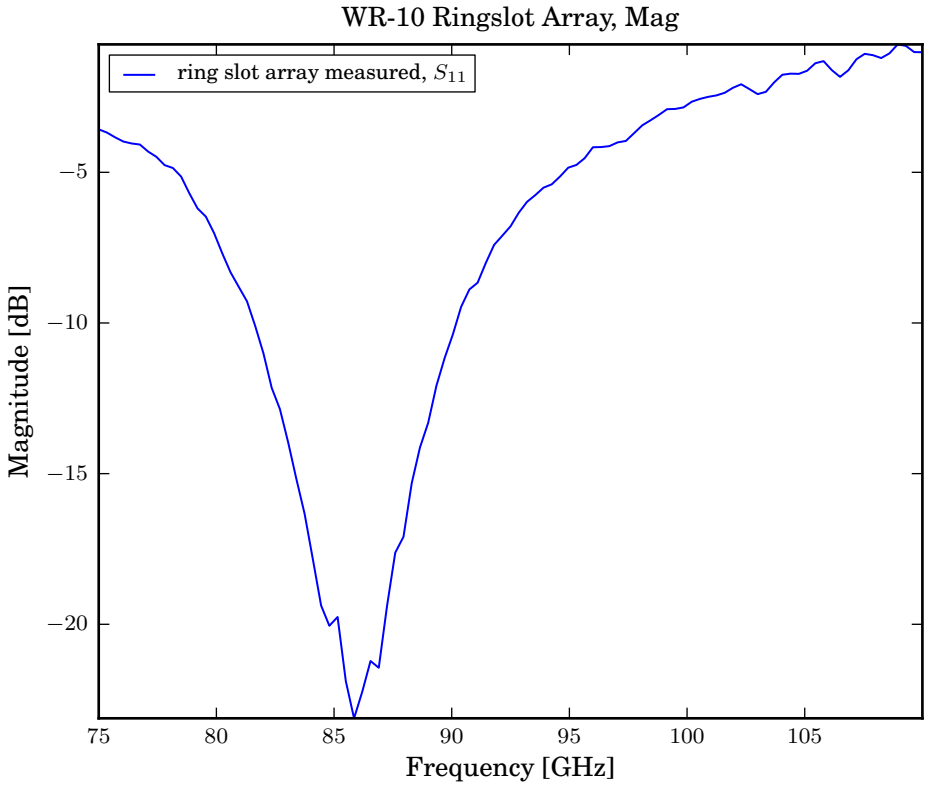
6.1 Basic Plotting

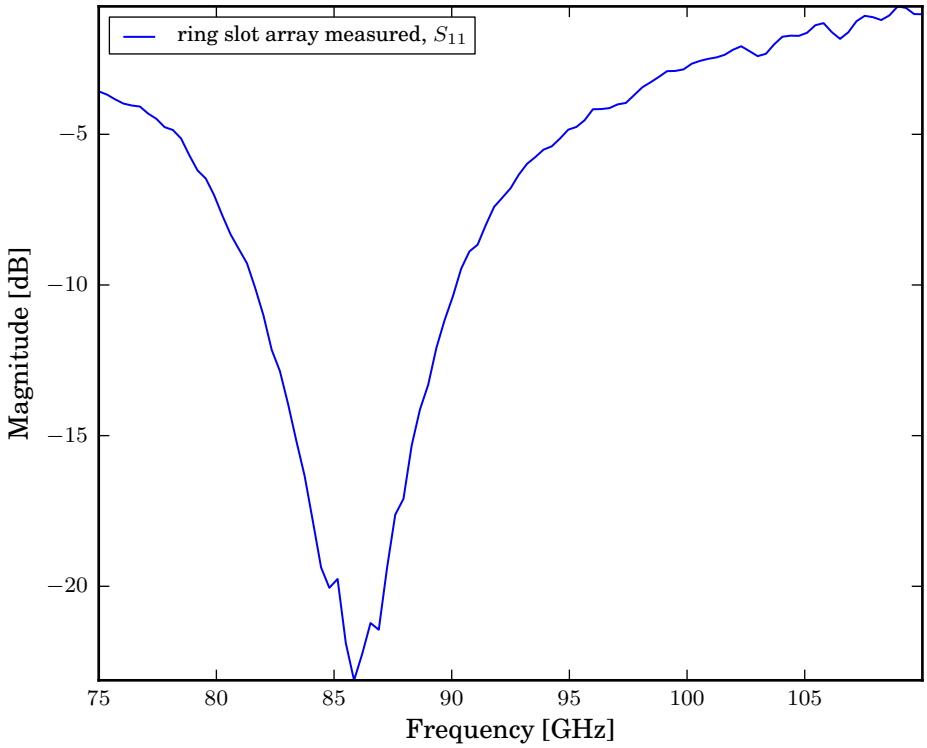
This example illustrates how to create common plots. The plotting functions are implemented as methods of the `Network` class, which is provided by the `mwavepy.network` module. Below is a brief list of the some commonly used plotting functions,

> Network.plot_s_smith Network.plot_s_complex Network.plot_s_db Network.plot_s_mag Network.plot_s_deg Network.plot_s_deg_unwrapped Network.plot_s_rad Network.plot_s_rad_unwrapped Network.plot_s_quad Network.plot_s_quad_unwrapped Network.plot_s_re Network.plot_s_im

6.1.1 Return Loss Magnitude

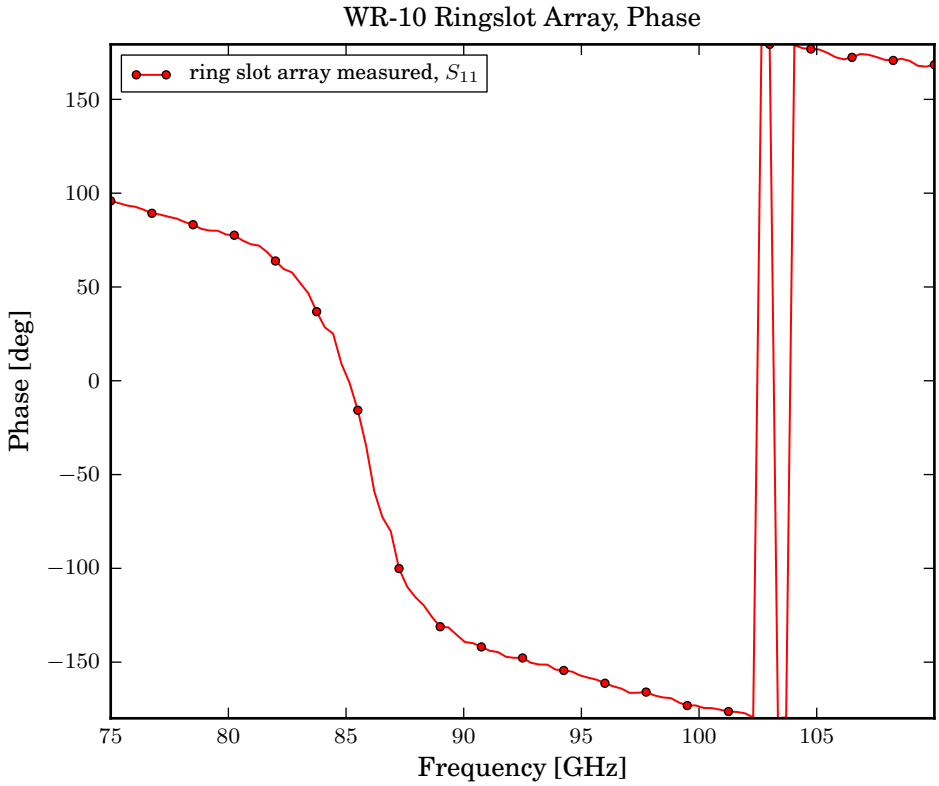
```
import pylab
import mwavepy as mv

# create a Network type from a touchstone file of a horn antenna
ring_slot= mv.Network('ring slot array measured.s1p')

# plot magnitude (in db) of S11
pylab.figure(1)
pylab.title('WR-10 Ringslot Array, Mag')
ring_slot.plot_s_db(m=0,n=0) # m,n are S-Matrix indecies
pylab.figure(2)
ring_slot.plot_s_db(m=0,n=0) # m,n are S-Matrix indecies
# show the plots
pylab.show()
```


WR-10 Ringslot Array, Mag
ring slot array measured, S_{11}
−5
−10
−15
−20
Magnitude [dB]
75
80
85
90
95
100
105
Frequency [GHz]



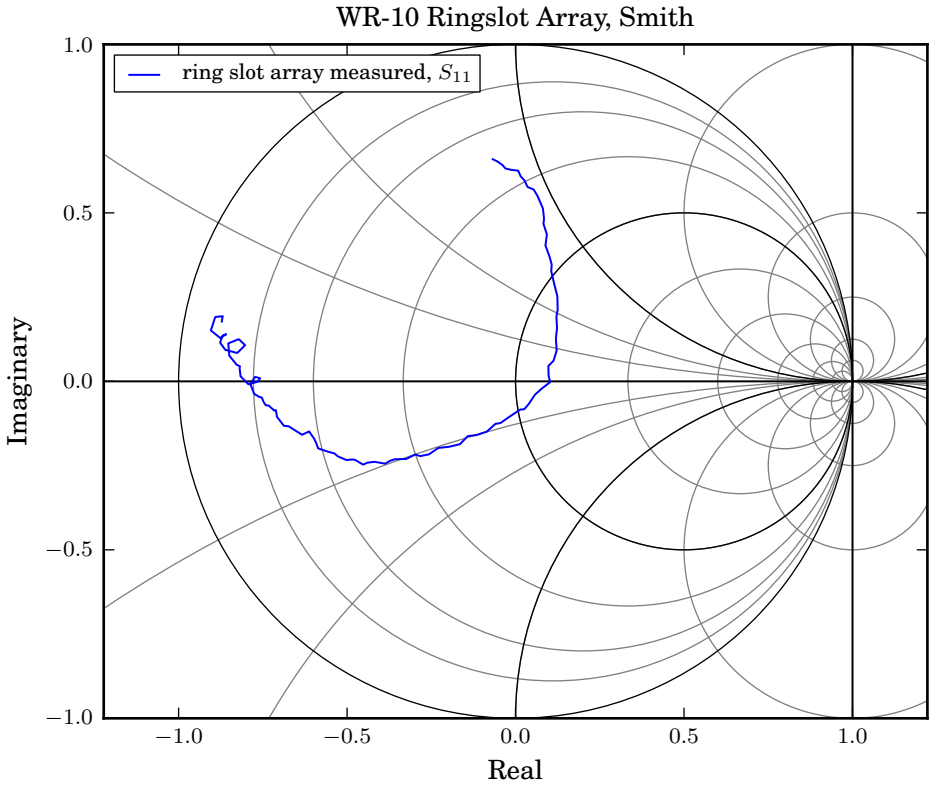
6.1.2 Return Loss Phase

```
import pylab
import mwavepy as mv

ring_slot= mv.Network('ring slot array measured.s1p')
pylab.figure(1)
pylab.title('WR-10 Ringslot Array, Phase')
# kwargs given to plot commands are passed through to the pylab.plot
# command
ring_slot.plot_s_deg(m=0,n=0, color='r', markevery=5, marker='o')
pylab.show()
```



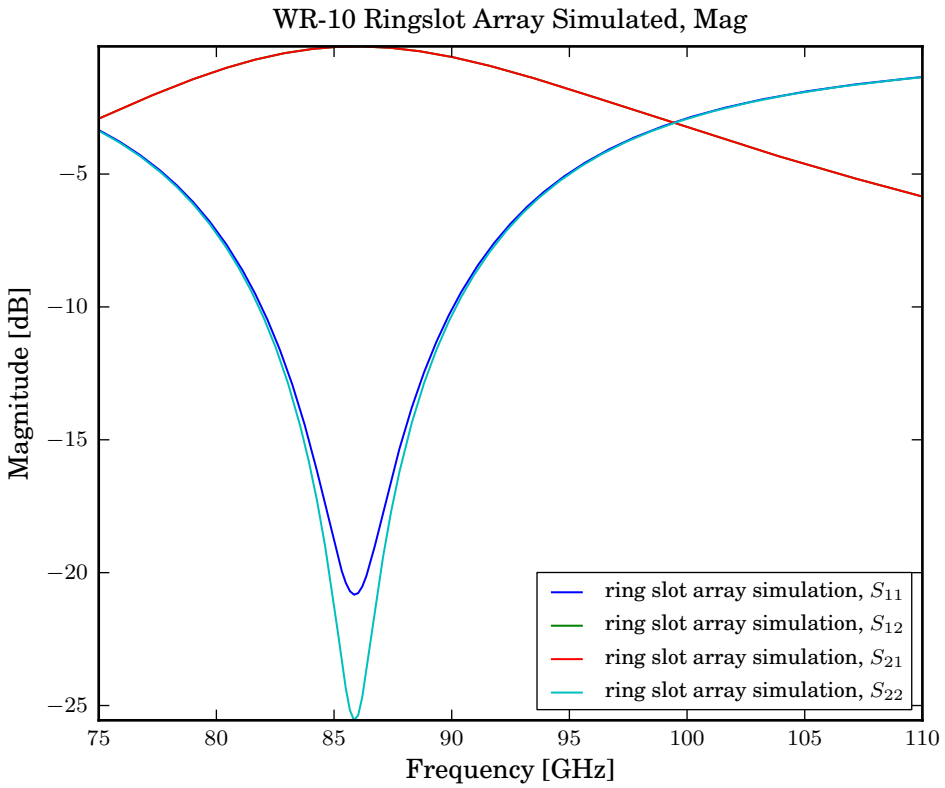
6.1.3 Return Loss Smith

```
import pylab
import mwavepy as mv

ring_slot= mv.Network('ring slot array measured.s1p')

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



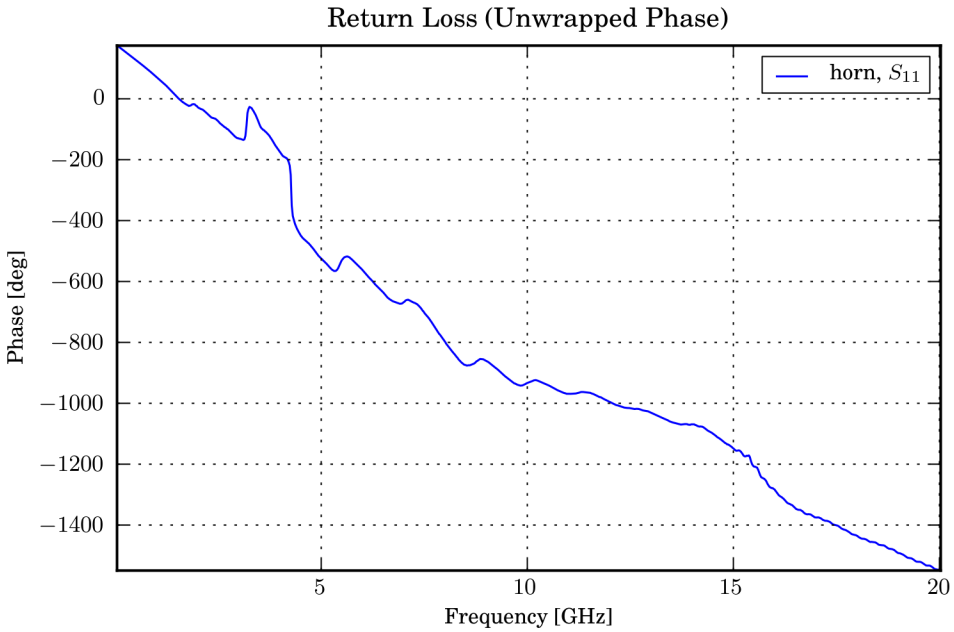
6.1.4 All S-parameters

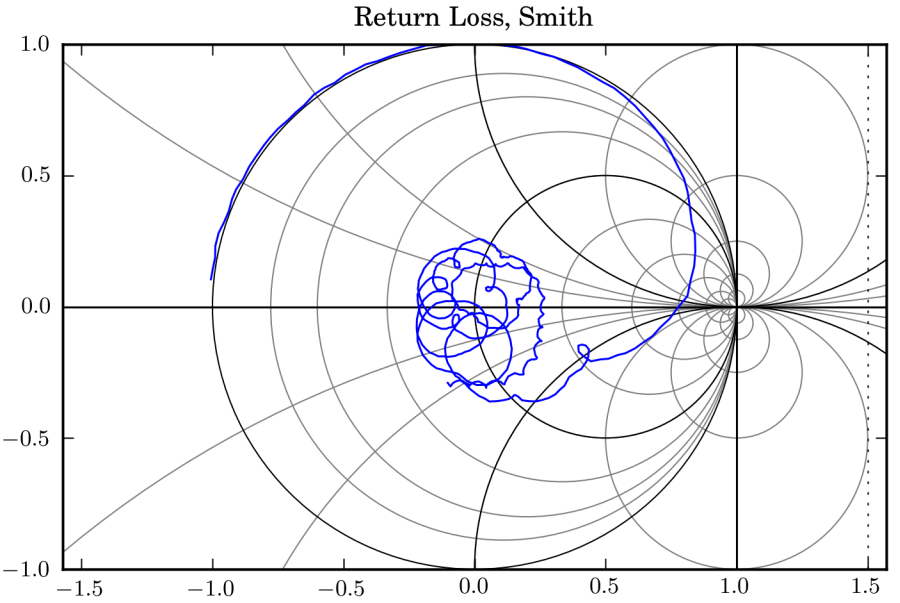
```
import pylab
import mwavepy as mv

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

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



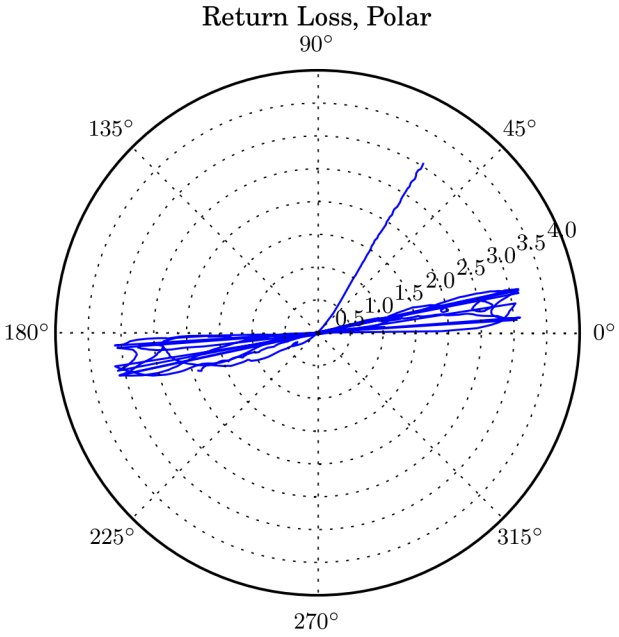
6.1.5 Comparing with Simulation

```
import pylab
import mwavepy as mv

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

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

# 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')
```


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

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




```
#  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

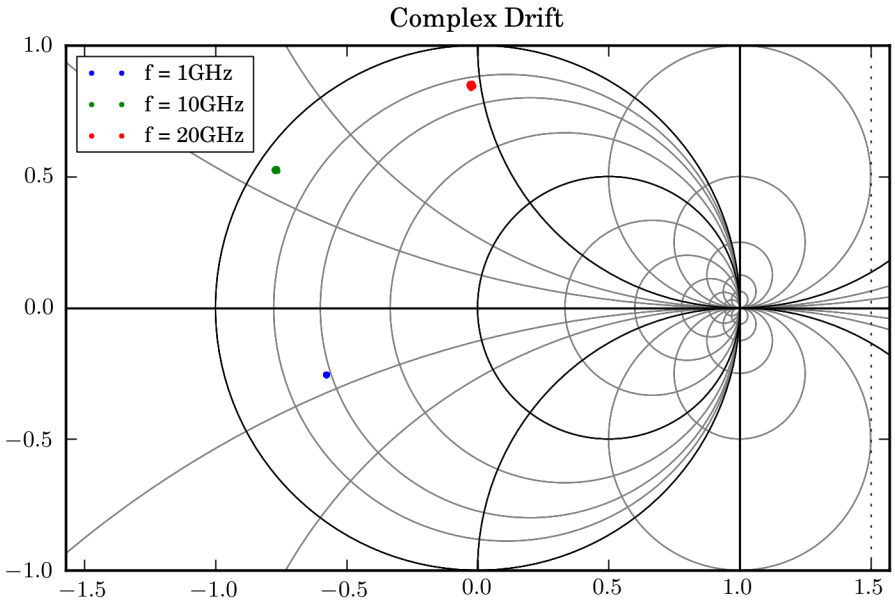
```
import mwavepy as mv
import os,datetime

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();
```



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)

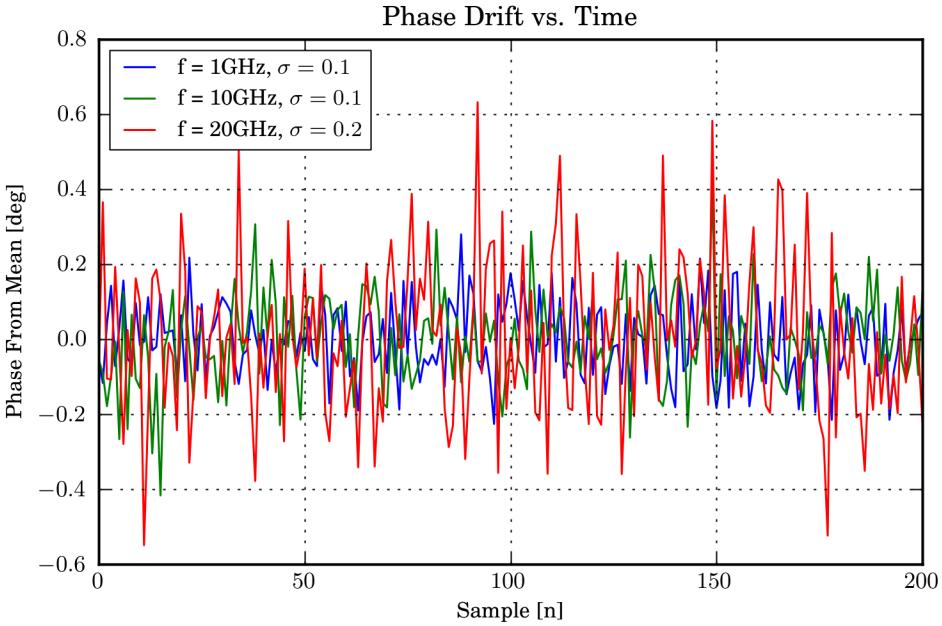
A single stub matching network can be designed to produce maximum power transfer to the load, at a single frequency. The matching network has two design parameters:

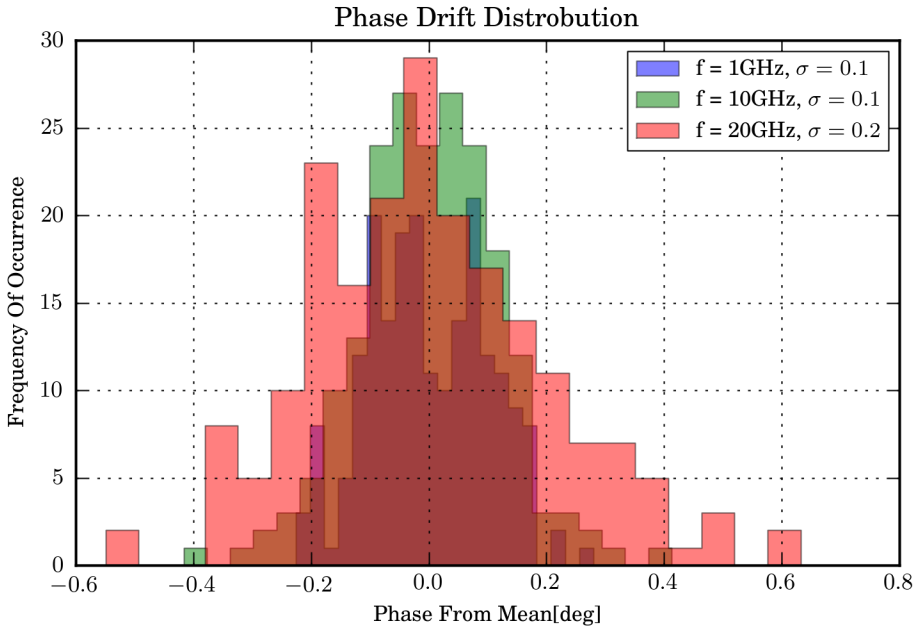
- length of series tline
- length of shunt tline

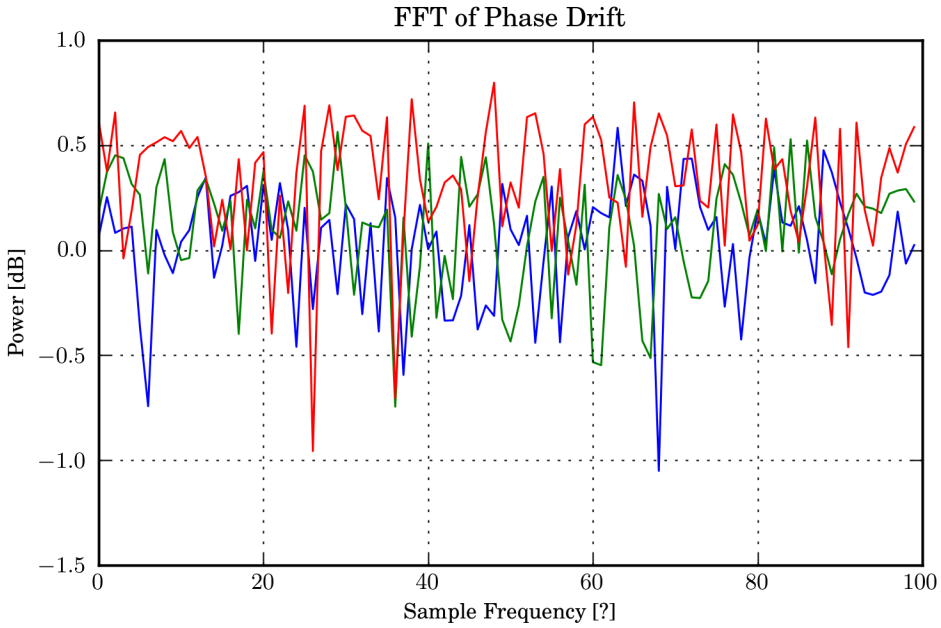
This script illustrates how to create a plot of return loss magnitude off the matched load, vs series and shunt line lengths. The optmial designs are then seen as the minima of a 2D surface.

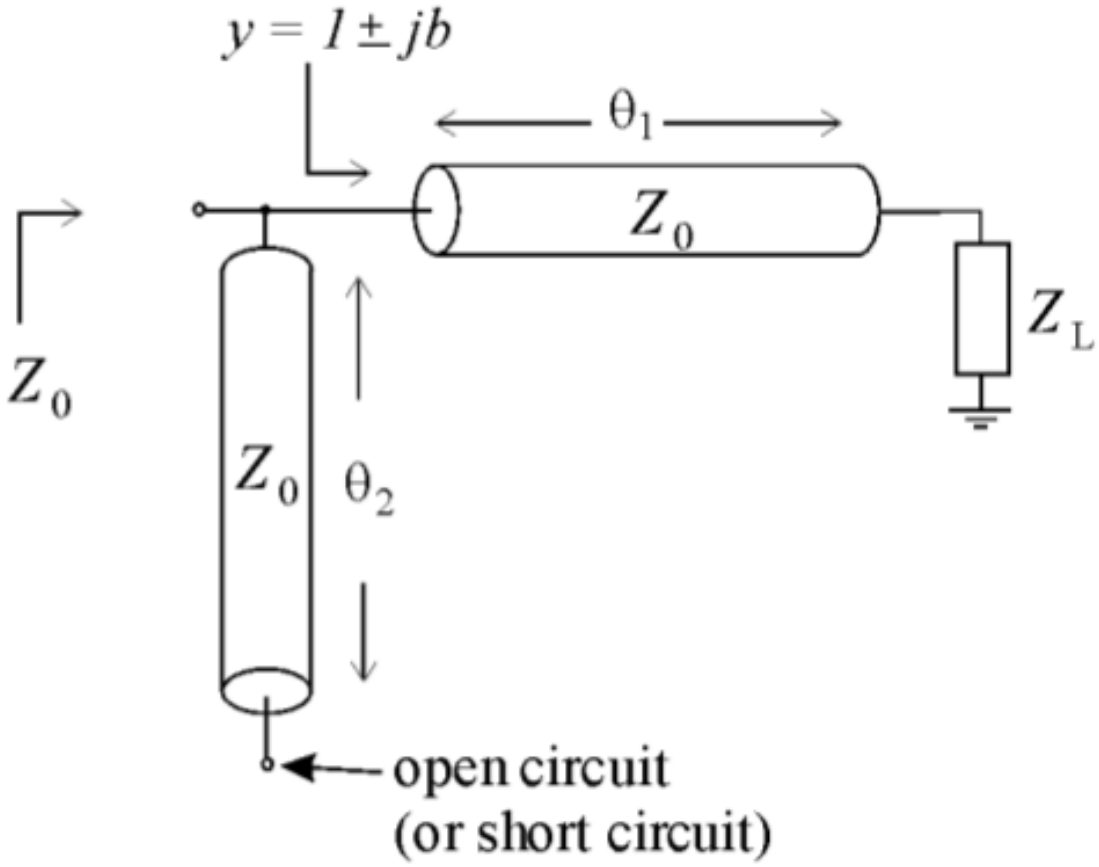
6.5.2 Script

```
import mwavepy as mv
from pylab import *


# Inputs
```


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


Phase Drift Distrobution
f = 1GHz, $\sigma = 0.1$
f = 10GHz, $\sigma = 0.1$
f = 20GHz, $\sigma = 0.2$
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

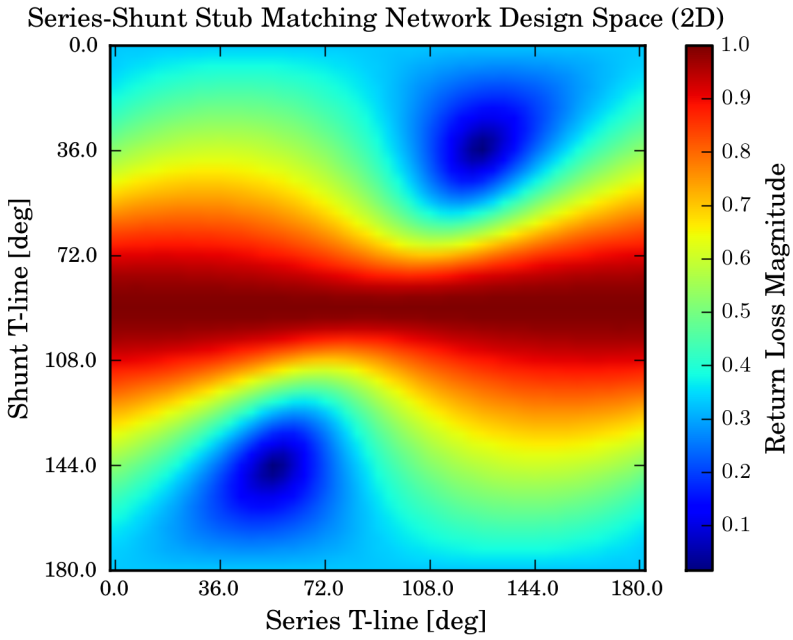

FFT of Phase Drift
Power [dB]
Sample Frequency [?]
1.0
0.5
0.0
−0.5
−1.0
−1.5
0
20
40
60
80
100

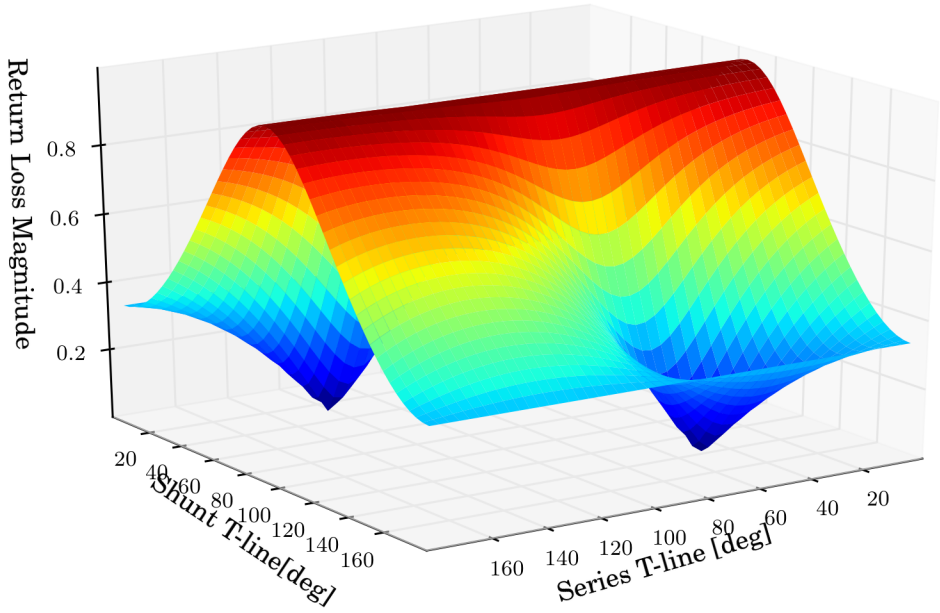

y = 1 ± jb
θ1
Z0
ZL
Z0
Z0
θ2
open circuit
(or short circuit)

```
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:


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


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

CHAPTER

SEVEN

NETWORK (MWAVEPY.NETWORK)

Provides a n-port Network class and associated functions.

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

7.1 Network Class

`Network`([touchstone_file, name])	An n-port microwave network.

7.1.1 mwavepy.network.Network

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

An n-port microwave network.

the most fundemental properties are:

- `s` : scattering matrix. a kxnxn complex matrix where 'n' is number of ports of network, and *k* is number of frequency points
- `z0`: characteristic impedance
- `f` : frequency vector in Hz. see `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 (see `connect()`)
- // : de-embdeding of one network from the other. (better to

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

Methods

add_noise_polar	
add_noise_polar_flatband	
change_frequency	
flip(a)	invert the ports of a networks s-matrix, 'flipping' it over
interpolate	
multiply_noise	
nudge	
plot_passivity	
plot_polar_generic	
plot_s_all_db	
plot_s_complex	
plot_s_db	
plot_s_deg	
plot_s_deg_unwrap	
plot_s_deg_unwrapped	
plot_s_im	
plot_s_mag	
plot_s_polar	
plot_s_quad	
plot_s_rad	
plot_s_rad_unwrapped	
plot_s_re	
plot_s_smith	
plot_vs_frequency_generic	
read_touchstone	
write_touchstone	

mwavepy.network.flip

mwavepy.network.**flip**(*a*)

invert the ports of a networks s-matrix, 'flipping' it over

note: only works for 2-ports at the moment

7.2 Functions On Networks

connect(ntwkA, k, ntwkB, l)	connect two n-port networks together.
innerconnect(ntwkA, k, l)	connect two ports of a single n-port network.
cascade(ntwkA, ntwkB)	cascade two 2-port Networks together
de_embed(ntwkA, ntwkB)	de-embed *ntwkA* from *ntwkB*. this calls *ntwkA.inv**ntwkB*.
average(list_of_networks)	calculates the average network from a list of Networks.
one_port_2_two_port(ntwk)	calculates the two-port network given a symetric, reciprocal and

7.2.1 mwavepy.network.connect

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.

Parameters **ntwkA** : `Network`

network 'A'

k : int

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

ntwkB : `Network`

network 'B'

l : int

port index on *ntwkB*

Returns **ntwkC** : `Network`

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

See Also:

connect_s actual S-parameter connection algorithm.

innerconnect_s actual S-parameter connection algorithm.

Notes

the effect of mis-matched port impedances is handled by inserting a 2-port 'mismatch' network between the two connected ports. This mismatch Network is calculated with the :func:impedance_mismatch function.

Examples

To implement a *cascade* of two networks

```
>>> ntwkA = mv.Network('ntwkA.s2p')
>>> ntwkB = mv.Network('ntwkB.s2p')
>>> ntwkC = mv.connect(ntwkA, 1, ntwkB,0)
```

7.2.2 mwavepy.network.innerconnect

`mwavepy.network.`**innerconnect**(*ntwkA*, *k*, *l*)

connect two ports of a single n-port network.

this results in a (n-2)-port network. remember port indecies start from 0.

Parameters **ntwkA** : `Network`

network 'A'

k : int

port index on ntwkA (port indecies start from 0)

l : int

port index on ntwkB

Returns **ntwkC** : Network

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

See Also:

connect_s actual S-parameter connection algorithm.

innerconnect_s actual S-parameter connection algorithm.

Notes

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

Examples

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

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

7.2.3 mwavepy.network.cascade

mwavepy.network.**cascade**(*ntwkA*, *ntwkB*)

cascade two 2-port Networks together

connects port 1 of *ntwkA* to port 0 of *ntwkB*. This calls *connect(ntwkA,1, ntwkB,0)*

Parameters **ntwkA** : Network

network *ntwkA*

ntwkB : Network

network *ntwkB*

Returns **C** : Network

the resultant network of ntwkA cascaded with ntwkB

See Also:

connect connects two Networks together at arbitrary ports.

7.2.4 mwavepy.network.de_embed

mwavepy.network.**de_embed**(*ntwkA*, *ntwkB*)

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

Parameters **ntwkA** : Network

network *ntwkA*

ntwkB : Network

network *ntwkB*

Returns **C** : Network

the resultant network of ntwkB de-embeded from ntwkA

See Also:

`connect` connects two Networks together at arbitrary ports.

7.2.5 mwavepy.network.average

`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

Parameters **list_of_networks: list** :

a list of `Network` objects

Returns **ntwk** : `Network`

the resultant averaged Network

Notes

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

Examples

```
>>> ntwk_list = [mv.Network('myntwk.s1p'), mv.Network('myntwk2.s1p')]
>>> mean_ntwk = mv.average(ntwk_list)
```

7.2.6 mwavepy.network.one_port_2_two_port

`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

7.3 Supporting Functions

`connect_s`(A, k, B, l)	connect two n-port networks' s-matricies together.
`innerconnect_s`(A, k, l)	connect two ports of a single n-port network's s-matrix.
`s2t`(s)	converts scattering parameters to scattering transfer parameters.
`t2s`(t)	converts scattering transfer parameters to scattering parameters
`inv`(s)	calculates inverse s-parameter matrix, used for de-embeding
`flip`(a)	invert the ports of a networks s-matrix, 'flipping' it over

7.3.1 mwavepy.network.connect_s

mwavepy.network.**connect_s**(*A*, *k*, *B*, *l*)

connect two n-port networks' s-matricies together.

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

Parameters **A** : numpy.ndarray

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

k : int

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

B : numpy.ndarray

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

l : int

port index on *B*

Returns **C** : numpy.ndarray

new S-parameter matrix

See Also:

connect operates on `Network` types

innerconnect_s function which implements the connection connection algorithm

Notes

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

7.3.2 mwavepy.network.innerconnect_s

mwavepy.network.**innerconnect_s**(*A*, *k*, *l*)

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

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

Parameters **A** : numpy.ndarray

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

k : int

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

l : int

port index on *A*

Returns **C** : numpy.ndarray

new S-parameter matrix

Notes

The algorithm used to calculate the resultant network is called a 'sub-network growth', can be found in [1]. The original paper describing the algorithm is given in [2].

References

7.3.3 mwavepy.network.s2t

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

converts scattering parameters to scattering transfer parameters.

transfer parameters [3] are also refered to as 'wave cascading matrix', this function only operates on 2-port networks.

Parameters **s** : numpy.ndarray

scattering parameter matrix. shape should be should be 2x2, or fx2x2

Returns **t** : numpy.ndarray

scattering transfer parameters (aka wave cascading matrix)

See Also:

t2s converts scattering transfer parameters to scattering parameters

inv calculates inverse s-parameters

References

7.3.4 mwavepy.network.t2s

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

converts scattering transfer parameters to scattering parameters

transfer parameters [4] are also refered to as 'wave cascading matrix', this function only operates on 2-port networks. this function only operates on 2-port scattering parameters.

Parameters **t** : numpy.ndarray

scattering transfer parameters, shape should be should be 2x2, or fx2x2

Returns **s** : numpy.ndarray

scattering parameter matrix.

See Also:

t2s converts scattering transfer parameters to scattering parameters

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

[2] Filipsson, Gunnar; , "A New General Computer Algorithm for S-Matrix Calculation of Interconnected Multiports," Microwave Conference, 1981. 11th European , vol., no., pp.700-704, 7-11 Sept. 1981. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4131699&isnumber=4131585

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

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

inv calculates inverse s-parameters

References

7.3.5 mwavepy.network.inv

mwavepy.network.**inv**(*s*)

calculates inverse s-parameter matrix, used for de-embeding

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

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

where x^{-1} is the matrix inverse.

Parameters **s** : numpy.ndarray

scattering parameter matrix. shape should be should be 2x2, or fx2x2

Returns **s'** : numpy.ndarray

inverse scattering parameter matrix.

See Also:

t2s converts scattering transfer parameters to scattering parameters

s2t converts scattering parameters to scattering transfer parameters

7.3.6 mwavepy.network.flip

mwavepy.network.**flip**(*a*)

invert the ports of a networks s-matrix, 'flipping' it over

note: only works for 2-ports at the moment

CHAPTER

EIGHT

FREQUENCY (MWAVEPY.FREQUENCY)

Provides a frequency object and related functions.

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

8.1 Frequency Class

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

8.1.1 mwavepy.frequency.Frequency

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

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.

Methods

`from_f`
`labelXAxis`

CHAPTER

NINE

INDICES AND TABLES

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

PYTHON MODULE INDEX

m

INDEX