API Documentation

API Documentation

October 19, 2016

Contents

1 Module UtilitiesLib

```
Created on Wed Jan 22 14:42:31 2014

@author: Oscar Gargiulo


This library is a colleciton of utilities functions, it also imports some phisical constants and the mos



Last changes:
v1.2.0 - OSC:
- renamed some function and added some help to them, removed some function related to some instruments
- added a function that load all the datamodules in a folder and eventually also one or more specified p

v1.1.6 - DAZ:
- added fuction to give segments to use for VNA measurement
    segments_vna(...)

v1.1.5 - DAZ:
- added function to read out information of complex dm
    multi_dm_cplx_readout()


v1.1.4 - oscar:
- added the function font_list()

v1.1.3 - DAZ:
- added function to import csv file containing freq., re and im data to complex
    datamodule (to replace fct added in v1.1.2)

v1.1.2 - DAZ:
- added function to combine two dat files (typ. from HFSS) to cplx. datamodule

v1.1.1 - DAZ :
- added function to combine real and imaginry datamodule to complex datamodule
- added function to combine several complex datamodule (typ. over diff. span)



v1.1.0:
the bins_creation function returns the bins
```

1.1 Functions

export_dat_file(*Filename*, *Separator*, *Header*, **args*)

```
function export_dat_file(Filename,Separator,Header,*args):

- The function write n column separated by 'Separator'. It is possible to pass
 a header.
The header can be a list of strings, one for each column.

Examples:

export_dat_file('test','        ','example1',x,y,z)

export_dat_file('test','        ',['colx','coly','colz'],x,y,z)

export_dat_file('test','        ',None,x,y,z)
```

import_dat_file(*filename*, *separator*, *skip_lines*, **args*)

```
function M=import_dat_file(filename,separator,skip_lines,*columns)

The function read a N column files separated by 'separator',return a matrix

Examples:
    1) The file has two lines of description, we want only the first and
    fifth column, the columns are separated by a tabulation:

        M=import_dat_file('fp.dat','\t',2,1,5)

    2) The file doesn't have description, we want all the file:

        M=import_dat_file('z:\my_files\file.dat','\t',0,'a')
```

export_dm_textfile(*data*, *Filename*, *Separator*='\t', *Header*='')

function export_dm_textfile(data,Filename,Separator=' ',Header='')

This function will create a text file where the columns contain the data from the data module.

default separator is tabulation, Header can be None or empty string, or text or a list/tuple with a string for each column

Examples: export_dm_textfile(data,'test',' ','example1')

export_dm_textfile(data,'test',' ',['colx','coly','colz'])

export_dm_textfile(data,'test',' ',None)

create_frequency_axis(*Time_axe*, *Mode*='m')

```
function f_axe=create_frequency_axe(Time_axe,[Mode='m'])

The function will evaluate the fourier space frequency axe using time_axe
as model

- Time_axe: array of numbers representing a signal time axe
- Mode: default is mirror, the frequency axe will be [-FS/2,FS/2]
where FS is the sampling frequence. The other mode is [0,FS], to change
mode just put anything but m or M as second argument

Examples:

- f_axe=create_frequency_axe(time_axe)
- f_axe=create_frequency_axe(time_axe,1)
```

create_time_axis(*Time*, *SF*=25)

function t_axe=create_time_axe(Time,[SF=25 (GHz)])

The function will create a time axe from 0 to Time, with a step equal to the sampling period 1/SF

bins_creation(*Measured*, *Expected*, *Binsnumber*, *Mode*=0, *Plot*=True, *Amp*=100)

```
function err,var=bins_creation(Measured,Expected,Binsnumber,[Mode=0],
    [Plot=True],[Amplitude=100 (%)])

Function used to create a distribution of the Amplitude and Phase measures
(array), the function will return the mean and variance. Moreover it will plot
the distribution

- Measured: is the array containing the measures
- Expected: is the array containing the expected values that every measure
should have or the single number that ALL measures should have
- Binsnumber: is the number of bins, or the resolution of the distribution,
the x-axis will be splittend in Binsnumber parts
- Plot: at the end of the elaboration it will display the bins (default=True)
- Amp: is the value of the amplitude, it is used only in the plot title
- Mode:

    -0 is for amplitude distribution
    -1 is for phase distribution
```

cut_pulse(*Signal*, *SF*, **args*)

```
function y1,y2,... = cut_pulse(Signal,SF (Gigasamples/s),*args (ns))

This function slice a sequence 'Signal' of pulses separated by a "wait time".

- FS is the sampling frequency of the time axe,
- Edge should be just after the noise
- args is the time position (in ns) of the wait time (it is necessary that it
is after one pulse and before the other one) if there is only one pulse it can
be omitted

- The output is a list of the size of args+1:
    Examples:
        1) with no args, the function cuts the wait time at the begin of
        the signal:

            - y=cut_pulse(Sig1,50,0.03)

        2) Two signals, wait time at the begin and between them, the first
        pulse is 100ns and the wait time is 5ns:

            - y1,y2=cut_pulse(Sig1_2,50,0.03,107)
```

cut_pulse_auto(*Signal*, *Edge*, *Minimum*=0, *smith_ist*=0, *Mode*='ZP')

```
This function slice a sequence 'Signal' of pulses separated by a
    "wait time".

    NOTE: there should be a wait time before and after the sequence, too



- Noise should be inside +Edge and -Edge

- The output is a list containing the number of pulses:

- Mode:
    - 'ZP': zero-padding, used to preserve the absolute phase, but not needed
    for a relative phase
    - 'C': cut, the pulse will be cutted, less memory used and faster

- Minimum: minimum number of samples of the cutted signal, helps to avoid noise
and bumps. Default=0


    Examples:
        1) list of pulses
            - y=cut_pulse(Sig1,0.02)
            first pulse: y[0]
            second pulse: y[1]
            ...
            last pulse y[len(y)-1]

        2) Separated pulse for a sequence of 3 pulses:

            - y1,y2,y3=cut_pulse(Sig1_3,0.04)
```

wait_time_measure(*Signal*, *SF*=25, *Edge*=0.03)

this function can be used to evaluate the waiting time at the begin of a signal, given the Edge (y-axis units)

load_scope_file(*Filename*, *Scope*, *Time_axis*=False)

```
This function will load a wave from a Matlab file saved with the scope
    in the default pc_folder shared with it.
The Wfm.dat extension will be added automatically.

- Scope: is the name of the variable containing the Scope initialized Class (Scope library)
- Time_axis: if it is True, the time axe will be generated and returned as first output (default=Fa
```

cut_wait_time(*Signal*, *Edge*=0.01)

This function is used to cat slices of a signal separated by a "wait time", given the Edge (y-axis units)

amplitude_phase(*Signal*, *Freq*, *Type*='s', *SF*=25, *Phaseref*=0, *Ampref*=1, *Sigma*=0)

```
-----------------------------------------------------------------

function a,p=amplitude_phase(Signal,Freq (GHz),[Type='s'],[SF=25 (Gigasamples/s)],[Phaseref=0 (rad)

The function evaluate the amplitude and phase difference of Signal comparing it with a wave create

- Freq is the frequency of the signal
- SF is the sampling frequency of the signal (25 Gigasamples/s default)
- Type:
    - 's': square wave AM (default)
    - 'g': gaussian wave AM
    - 't': triangle wave AM

- Sigma: only in case of a gaussian type, sigma can be inserted (time/6 as default)
```

float_to_hexstring(*Number*)

this function converts a float number to an hex number in string format

time_difference(*Reference*, *Multiplier*=1)

given a time as a reference, this function will evaluate how much time is passed since the reference time.

It is possible to pass a Multiplier (def 1), the final time is multiplied by it

shift90(*Sig*, *Sig_freq*, *Sampling_freq*, *Truncated*=True)

```
function Sig_out,Sig_out_90 = shift90(Sig,Sig_freq (MHz),Sampling_freq (GHz),Truncated=True):

This function will shift a signal of 90 degrees, as example a cosin will become a sin.

- Sig is the original signal
- Sig_freq (MHz) is the signal frequency and Sampling_freq (GHz) is the sampling frequency: they ar
- Truncated (def:True): is used to truncate the output signals so that their length will be equal
                        if it is False, a zero padding will be added at the end of the outputs to

NOTE: to have an integer number the Sig_freq (MHz) should be a multiple of 250*Sampling_freq (GHz)
```

IQ_average(*Signal*, *Reference*, *Sampling_freq*, *DM_freq*, *Discard_start*=0, *Discard_stop*='L')

```
Given a list of signals and references, this function will perform the
IQ decomposition of each signal, using the given reference.
Then the average of I(t) and Q(t) will be performed

Parameters:
- Sampling_freq (Gigasamples/s) is the SF used to sample the signals and
references
- DM_freq (MHz) is the frequency of the acquired signal and reference
- Discard_start (ns) is used to discard the initial part of the signal,
useful if there is a delay (DEF is 0)
- Discard_stop (ns) is used to discard the final part of the signal,
(DEF is 'L', no discard)
```

IQ2PHAM(*I*, *Q*)

This function converts I and Q params in phase and amplitude

S2PRead(*input_name*, *plots_name*='', *save_plots*=True, *save_data*=True, *Simmetric*=True, *return_all*=False)

```
This function reads a S2P files saved on the VNA and return the S
parameters, plotting and saving them.

arguments:

- input_name is the name of the file
- plots_name is the name of the plots if the option to save them is True,
the name of the plots will be followed by '- Sxx'
- save_plots can be 1 or True, 0 or False, it is used to save the plots
- Simmetric: if this value is True than S21 is ignored (it is equal to S12)
- return_all the function will return all 8 parameters
(S parameter + calibration par)
```

filter_data(*y*, *window_size*, *order*, *deriv*=0, *rate*=1)

```
Smooth (and optionally differentiate) data with a Savitzky-Golay filter.
The Savitzky-Golay filter removes high frequency noise from data.
It has the advantage of preserving the original shape and
features of the signal better than other types of filtering
approaches, such as moving averages techniques.
Parameters
----------
y : array_like, shape (N,)
    the values of the time history of the signal.
window_size : int
    the length of the window. Must be an odd integer number.
order : int
    the order of the polynomial used in the filtering.
    Must be less then ‘window_size‘ - 1.
deriv: int
    the order of the derivative to compute (default = 0 means only
    smoothing)
Returns
-------
ys : ndarray, shape (N)
    the smoothed signal (or it’s n-th derivative).
Notes
-----
The Savitzky-Golay is a type of low-pass filter, particularly
suited for smoothing noisy data. The main idea behind this
approach is to make for each point a least-square fit with a
polynomial of high order over a odd-sized window centered at
the point.
Examples
--------
t = np.linspace(-4, 4, 500)
y = np.exp( -t**2 ) + np.random.normal(0, 0.05, t.shape)
ysg = savitzky_golay(y, window_size=31, order=4)
import matplotlib.pyplot as plt
plt.plot(t, y, label=’Noisy signal’)
plt.plot(t, np.exp(-t**2), ’k’, lw=1.5, label=’Original signal’)
plt.plot(t, ysg, ’r’, label=’Filtered signal’)
plt.legend()
plt.show()
References
----------
.. [1] A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of
   Data by Simplified Least Squares Procedures. Analytical
   Chemistry, 1964, 36 (8), pp 1627-1639.
.. [2] Numerical Recipes 3rd Edition: The Art of Scientific Computing
   W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery
   Cambridge University Press ISBN-13: 9780521880688
```

write_waves_csv(*filename*, *wavename*, *data*, *type*=`'ANALOG_16'`)

```
funtion write_waves_csv(filename,wavename,data,[type='ANALOG_16'])

This function creates a csv file that can be loaded in the AWG or DIO
memory

types are:
- ANALOG_16
- ANALOG_32
- ANALOG_16_DUAL
- ANALOG_32_DUAL
- IQ
- IQPOLAR
- DIGITAL
```

progressive_plot_2d(*x*, *y*, *style*=`'b'`, *clear*=`True`)

This function can be used to make a progressive plot, everytime the x and y axis are updated, the function must be called again

progressive_plot_3d(*x*, *y*, *z*, *Levels*=`10`)

This function can be used to make a progressive plot, everytime the x and y axis are updated, the function must be called again

This function uses contourf

combine_re_im_cplx(*re*, *im*)

Combines 2d data modolue one containing real other imaginary data

combine_re_im_dat_file_cplx(*loc_re*, *loc_im*)

As .csv filess should be exported this function should not be used any more. It remains to ensure downwards compatibility. This function combines two .dat files each containg frequency in the first column and real respectivly complex part of the S parameter in the second column. Expected arguments are the location of those dat files Returns complex datamodule

import_csv_to_cplx_dm(*file_name*, *print_header*=`False`)

Creates complex data module from .csv file. First column MUST BE freq. second corresponding real and third imaginary data (as expected from HFSS). Furthermore it checks the corresponding table headers (and gives warnings), and stores the headers of the table in comments of datamodule.

multi_dm_cplx_readout(*data*)

Functions gets array of complex data modules and returns array of: Qi, Qi_err, Qc, Qc_err, Ql, Ql_err, fr, fr_err

segments_vna(*centerfrq*, *span_tot*, *span_detail*, *npoints_wings*, *npoints_detail*)

Returns array of three dictionaries to hand over to VNA. parameters: centerfrq (GHz), span_tot (GHz) –> begin meas centerfrq-span/2 span_detailed (GHz) span which is measured in detail (with npoints_detail) npoints_wings: number of points used on each of the wings

font_list()

returns the font list names that can be used in maplotlib.pyplot

load_datamodules_in_folder(*path*='.', *parameters*=[])

This function opens all the datamodules in a folder and returns them in a list

It is possible to pass a list that contains parameters name, new lists containing the specified parameters will be returned. If the parameter name is not in the datamodule, an error will occur.

Examples:

dmlist = load_datamodules_in_folder('test')

dmlist, powers = load_datamodules_in_folder('test',['excitation_power']) powers = powers[0]

dmlist, powers = load_datamodules_in_folder('test', ['excitation_power','ex_frequency']) freqs = powers[1] powers = powers[0]

1.2 Variables

Name	Description
electron_mass	**Value:** `9.1093826e-31`
e_charge	**Value:** `1.60217653e-19`
hbar	**Value:** `1.054571726e-34`
speed_of_light	**Value:** `299792458`
kb	**Value:** `1.3806488e-23`
n_avogadro	**Value:** `6.02214129e+23`
__package__	**Value:** `None`

1.3 Class ListTable

Overridden list class which takes a 2-dimensional list of the form [[1,2,3],[4,5,6]], and renders an HTML Table in IPython Notebook.

1.3.1 Methods

_repr_html_(*self*)

Inherited from list

__add__(), __contains__(), __delitem__(), __delslice__(), __eq__(), __ge__(), __getattribute__(), __getitem__(), __getslice__(), __gt__(), __iadd__(), __imul__(), __init__(), __iter__(), __le__(), __len__(), __lt__(), __mul__(), __ne__(), __new__(), __repr__(), __reversed__(), __rmul__(), __setitem__(), __setslice__(), __sizeof__(), append(), count(), extend(), index(), insert(), pop(), remove(), reverse(), sort()

Inherited from object

__delattr__(), __format__(), __reduce__(), __reduce_ex__(), __setattr__(), __str__(), __subclasshook__()

1.3.2 Properties

Name	Description
Inherited from object	
__class__	

1.3.3 Class Variables

Name	Description
Inherited from list	
__hash__	

Index