```python
# -*- coding: utf-8 -*-
"""
Created on Wed May 18 14:18:39 2017

@author: mullerrs

RUN SCRIPT IN SAME DIRECTORY AS ALL .dat DOCUMENTS OF THE DOM MEASUREMENTS
INCLUDING THE ELECTRONICS .wav FILE OF THE NOISE MEASUREMENT, AND
INCLUDING THE REFERENCE HYDROPHONE .dat MEASUREMENT TO CALIBRATE WITH
I'm sorry I didn't made this automatic yet.

WHEN RUNNING SCRIPT IN COMMANDLINE IT DOES NOT NEED ANY ARGUMENTS (yet)

I also want to implement a -h function and other parsers...
but first things first.

"""
#==============================================================================
# IMPORT LIBRARIES, FUNCTIONS AND CLASSES
#==============================================================================

import sys
import numpy as np
import struct
import os
import math
import scipy.io
import matplotlib.pyplot as plt
from datetime import datetime
from scipy.signal import butter, lfilter
import wavio
from numpy.linalg import norm as Norm
from numpy.fft import fft

#==============================================================================
# GLOBAL VARIABLES
#==============================================================================

audioFs      = 15.25 # frequency of audio source pulses

save_txt     = False # if save_txt == True --> all txt files, are being saved
plotjes      = False # if plotjes == True --> plots are being made           <==
save_png     = False # if save_png == True --> all png files, are being saved
show         = False # if show == True --> all plot's are being showed
cumm         = False # if cumm == Trus --> cumpute cummulatives. <== also not always
#==============================================================================
# FILTER
#==============================================================================

def butter_lowpass(cutoff, Fs, order=5):
    nyq = 0.5 * Fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    return b, a

def butter_lowpass_filter(data, basename, cutoff, Fs, order=5):
    b, a = butter_lowpass(cutoff, Fs, order=order)
    y = lfilter(b, a, data)
    basename = basename+'_lp%s' %(cutoff/1000)
    return y, basename
```

```python
def butter_highpass(cutoff, Fs, order=5):
    nyq = 0.5 * Fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype='high', analog=False)
    return b, a

def butter_highpass_filter(data, basename, cutoff, Fs, order=5):
    b, a = butter_highpass(cutoff, Fs, order=order)
    y = lfilter(b, a, data)
    basename = basename+'_hp%s' %(cutoff/1000)
    return y, basename

def butter_bandpass_filter(data, basename, lowcut, highcut, Fs, order=5):
    y = butter_lowpass_filter(data, basename, highcut, Fs, order)[0]
    basename = basename+'_bp%s-%s' %(lowcut/1000, highcut/1000)
    print basename
    return butter_highpass_filter(y, basename, lowcut, Fs, order)[0], basename

#==============================================================================
# CUT
#==============================================================================

def Cut(data, samples, Fs, basename):                                          # <=

    MAX = np.max(data)                  # Maximum valule in dataset
    treshold = MAX/2.5
    WHERE = np.where(data > treshold)  # Samples with value above treshold
    WHERE = np.asarray(WHERE[0])        # Make an array of the tuple
    start_pulse = [WHERE[0]]            # Create an array start_pulse & put
                                        # first sample above MAX/2 init
    # In de komende loop: check voor alle samples boven MAX/2 of ze bij
    # dezelfde pulse horen, of een nieuwe pulse zijn definitie nieuwe pulse is
    # als twee opeenvolgende samples met waarde boven treshold, meer dan Fs/16
    # van elkaar af liggen als een nieuwe pulse gevonden is: voeg deze toe aan
    # start_pulse
    i = 1
    while i < len(WHERE):
        if WHERE[i] > (start_pulse[-1] + (Fs/(int(audioFs)+1))):
            start_pulse.append(WHERE[i])
        i += 1

    N = samples
    cutted_data = np.array([])
    i = 0
    j = 0 # <- to count the removed pulses
    while i < len(start_pulse):
        range_I = start_pulse[i] - N/2
        cutted_pulse = (data[range_I : range_I+N])    #NB: hij heeft links 1
        # sample meer dan rechts van start_pulse. : is TOT, niet t/m.

        if range_I < 0:
            print 'IndexError occurred - Nothing to worry about: Skipped pulse'
            i += 1
            j += 1
        elif range_I+N > len(data):
            print 'IndexError occurred - Nothing to worry about: Skipped pulse'
            i += 1
            j += 1
        else:
```

```python
            cutted_data = np.concatenate((cutted_data, cutted_pulse))
            i += 1

    basename = basename + '_pulsecut%s' %samples

    x = float(len(cutted_data))/float(N) #how often 1024 samples in cutted data
    if x != len(start_pulse) - j:
        print 'Not all pulses are cutted correctly'
        print '\n Extra info to search for error: \n'
        print 'Expected amount of pulses from data: int of:       ', len(data)/Fs*auc
        print 'Amount of pulses that should have been plotted =  ', len(start_pulse)
        print 'Amount of pulses cutted =                           ', x
        print 'Max value                                           ', MAX
        print 'Treshold value                                      ', MAX/treshold
        print 'Start of the pulses that should have been plotted: \n', start_pulse

    ####### PLOT DATA #######
    if plotjes == True:
        l = N      # unit of file to read in:  [raw: Fs],   [cut: 1024]
        x = np.linspace(0, l-1, l)
        plt.plot(x, cutted_data[0:l], linewidth=1)
        plt.tick_params(axis='both', labelsize=15)
        if '_wavelet_' in basename:
            HYD = 'DOM piezo'
        else:
            HYD = 'reference hydrophone'
        plt.title('\n Single pulse of %s' %HYD, fontsize=24)
        plt.suptitle('used: %s'%basename, fontsize=9)
        plt.xlabel('# samples', fontsize=18)
        plt.ylabel('amp', fontsize=18)
        plt.tight_layout()
        plt.grid()
        plt.grid('on','minor')
        if save_png == True:
            plt.savefig(basename+'_data_one-pulse.png') #_data_1wav_zoom.png
        if show == True:
            plt.show()
        plt.clf()
    #########################

    return cutted_data, basename

#==============================================================================
# Make time span
#==============================================================================

def MakeTimeSpan(length, Fs):
    return np.linspace(0, length-1, length)/Fs

#==============================================================================
# Make frequency span
#==============================================================================

def MakeFreqSpan(Fs, NFFT):
    return np.linspace(0, NFFT/2-1, NFFT/2)*float(Fs)/float(NFFT)

#==============================================================================
# based on Ed's script
#==============================================================================
def plot(data, basename, Fs, NFFT):
```

```
    length  = np.size(data)
    length  = len(data)
    time    = MakeTimeSpan(length, Fs)

    plt.plot(time, data)
    stdev = (np.std(data))
    str = "$\sigma$: %6.4f Pa" % (stdev);
    plt.tick_params(axis='both', labelsize=15)
    plt.title('\n\nTimetrace', fontsize=24)
    plt.suptitle('used: %s\nNFFT: %s\n' %(basename, NFFT) + str, fontsize=9)
    str = "%s -> [%s] " %('amplitude', 'Pa');
    plt.ylabel(str, fontsize=18)
    plt.autoscale(tight = True)
    plt.xlabel('time ->[s]', fontsize=18)
    plt.grid()
    plt.grid('on', 'minor')
    plt.tight_layout()
    if save_png == True:
        plt.savefig(basename+'.png')
    if show == True:
        plt.show()
    plt.clf()

########################                                                    #<==

def MakeSpectrum(Fs, NFFT, data, basename):

    length      = len(data)
    PSD         = np.zeros((NFFT/2, 1), dtype=float)
    freq        = MakeFreqSpan(Fs, NFFT)
    Segment     = int(length/NFFT)
    if 'sweep' in basename:
        wnd     = np.ones(NFFT)
        basename = basename+'_wnd_rect'
        print '    rectangular window used'
    else:
        wnd     = np.hanning(NFFT)
        basename = basename+'_wnd_hann'
        print '    hanning window used'
    norm        = Norm(wnd)**2
    double2single    = 2.0
    for span in range(0, Segment):
        Bg                  = span*NFFT
        end                 = Bg+NFFT
        yw                  = wnd*data[Bg:end]
        a                   = fft(yw, NFFT)
        ac                  = np.conj(a)
        pxx                 = np.abs(a*ac)
        PSD[:, 0]          +=  double2single*pxx[0:NFFT/2]
    PSD[:, 0]  /= (float(Segment)*NFFT*norm)                                #<==
    basename = basename+ '_spectrum'
    return PSD[:, 0], basename

########################

def PlotSpectrum(Fs, NFFT, data, basename):

    PSD       = data

    if cumm == True:
```

```
        if "10-60kHz" in basename:
            FcumBegin    = 10e3 - 5e3
            FcumEnd      = 60e3 + 5e3
        elif "50_80kHz" in basename:
            FcumBegin    = 50e3 - 5e3
            FcumEnd      = 80e3 + 5e3
        elif "Sw50to80kHz" in basename:
            FcumBegin    = 50e3 - 5e3
            FcumEnd      = 80e3 + 5e3
        elif "DOM_40_ruis" in basename:
            FcumBegin    = 30e3 - 5e3
            FcumEnd      = 30e3 + 5e3
        elif "2kHz" in basename:
            FcumBegin    = 2e3 - 5e3
            FcumEnd      = 2e3 + 5e3
        elif "_10kHz" in basename:
    #         FcumBegin    = 10e3 - 5e3
    #         FcumEnd      = 10e3 + 5e3
            FcumBegin    = 3500
            FcumEnd      = 22500
        elif "_20kHz" in basename:
            FcumBegin    = 20e3 - 5e3
            FcumEnd      = 20e3 + 5e3
        elif "_30kHz" in basename:
            FcumBegin    = 30e3 - 5e3
            FcumEnd      = 30e3 + 5e3
        elif "_40kHz" in basename:
            FcumBegin    = 40e3 - 5e3
            FcumEnd      = 40e3 + 5e3
        elif "_50kHz" in basename:
            FcumBegin    = 50e3 - 5e3
            FcumEnd      = 50e3 + 5e3
        elif "_60kHz" in basename:
            FcumBegin    = 60e3 - 5e3
            FcumEnd      = 60e3 + 5e3
        else:
            FcumBegin    = 3500
            FcumEnd      = Fs/2

        dF         = float(Fs)/float(NFFT)     # amount of freq per bin
        idxB       = int(FcumBegin/dF)         # bin to start cumsum
        idxE       = int(FcumEnd/dF)           # bin to stop cumsum
        cummulative = PSD[idxB:idxE].cumsum();
        std = 10*np.log10(cummulative[-1]); #According to Parseval
    else:
        std = ''
    freq = MakeFreqSpan(Fs, NFFT)

################################################
#    Plot sqrt(PSD), dus met Pa op y-as
################################################
#    sqrt_std = np.sqrt(cummulative[-1])
#    plt.plot(freq, np.sqrt(abs(PSD)), zorder = 1, linewidth=1) # PSD in dB re amp^2
#    #plot cummulatives
#    plt.plot(freq[idxB:idxE], abs(cummulative), linewidth=1)
#    plt.plot(freq[idxE:idxB-1:-1], abs(PSD[idxE:idxB-1:-1].cumsum()), linewidth=1)
#    plt.tick_params(axis='both', labelsize=15)
#    str = "$\sigma$ = %6.4f Pa" % (sqrt_std)
#    plt.title('\n\nPowerspectrum', fontsize=24)
#    plt.suptitle('used: %s\nNFFT: %s\n %s' %(basename, NFFT, str), fontsize=9)
```

```
#    plt.ylabel("ampl [Pa] ", fontsize=18)
#    plt.autoscale(tight = True)
#    plt.grid()
#    plt.grid('on', 'minor')
#    plt.xlabel('Frequency [Hz]', fontsize=18)
#    plt.tight_layout()
#    if save_png == True:
#        plt.savefig(basename+'.png')
#    if show == True:
#        plt.show()
#    plt.clf()

#################################################
#   Plot 10*log10(PSD), dus dB re Pa op y-as
#################################################
    plt.plot(freq, 10.*np.log10(abs(PSD)), linewidth=1) # PSD in dB re amp^2
    #plot cummulatives
    if cumm == True:
        plt.plot(freq[idxB:idxE],10.*np.log10(abs(cummulative)), linewidth=1)
        plt.plot(freq[idxE:idxB-1:-1], 10.*np.log10(abs(PSD[idxE:idxB-1:-1].cumsum()
    plt.tick_params(axis='both', labelsize=15)
    if cumm == True:
        str = "$\sigma$ = %6.4f dB re Pa" % (std)
    else:
        str = ''
    plt.title('\n\nPowerspectrum', fontsize=24)
    plt.suptitle('used: %s\nNFFT: %s\n %s' %(basename, NFFT, str), fontsize=9)
    plt.ylabel("dB re Pa", fontsize=18)
    #plt.autoscale('y', tight = True)
    plt.grid()
    plt.grid('on', 'minor')
    plt.xlabel('Frequency [Hz]', fontsize=18)
    plt.xlim(3000, 88000)
    plt.tight_layout()
    if save_png == True:
        plt.savefig(basename+'_dBrePa.png')
    if show == True:
        plt.show()
    plt.clf()

#################################################
#   Plot 10*log10(PSD), dus dB re microPa op y-as
#################################################

    if cumm == True:
        std = 10*np.log10(cummulative[-1]/(1e-6)**2);
    plt.plot(freq, 20*np.log10(np.sqrt(abs(PSD))/1e-6), linewidth=1) # dB re $\mu$Pa
    #plot cummulatives
    if cumm == True:
        plt.plot(freq[idxB:idxE],10.*np.log10(abs(cummulative)/(1e-6)**2), linewidth
        plt.plot(freq[idxE:idxB-1:-1], 10.*np.log10(abs(PSD[idxE:idxB-1:-1].cumsum()
    plt.tick_params(axis='both', labelsize=15)
    if cumm == True:
        str = "$\sigma$ = %6.4f dB re Pa" % (std)
    else:
        str = ''
    plt.title('\n\nPowerspectrum', fontsize=24)
    plt.suptitle('used: %s\nNFFT: %s\n %s' %(basename, NFFT, str), fontsize=9)
    plt.ylabel("dB re $\mu$Pa", fontsize=18)
    #plt.autoscale('y', tight = True)
```

```python
    plt.grid()
    plt.grid('on', 'minor')
    plt.xlabel('Frequency [Hz]', fontsize=18)
    plt.xlim(3000, 88000)
    plt.tight_layout()
    if save_png == True:
        plt.savefig(basename+'_dBremuPa.png')
    if show == True:
        plt.show()
    plt.clf()

    return

#==============================================================================
# PSD Ed - for calibration
#==============================================================================

def Spectrum(Fs, NFFT, data, basename):

    length      = len(data)
    PSD         = np.zeros((NFFT/2, 1), dtype=float)
    freq        = MakeFreqSpan(Fs, NFFT)
    Segment     = int(length/NFFT)
    wnd         = np.hanning(NFFT)
    norm        = Norm(wnd)**2
    double2single    = 2.0
    for span in range(0, Segment):
        Bg                  = span*NFFT
        end                 = Bg+NFFT
        yw                  = wnd*data[Bg:end]
        a                   = fft(yw, NFFT)
        ac                  = np.conj(a)
        pxx                 = np.abs(a*ac)
        PSD[:, 0]          +=  double2single*pxx[0:NFFT/2]
    PSD[:, 0]  /= (float(Segment)*NFFT*norm)

    if plotjes == True:

        plt.plot(freq, PSD[:,0], linewidth=1)
        plt.tick_params(axis='both', labelsize=15)
        plt.title('\nPowerspectrum', fontsize=24)
        plt.suptitle('used: %s\nNFFT: %s'%(basename, NFFT), fontsize=9)
        plt.xlabel('Frequency [Hz]', fontsize=18)
        plt.ylabel('amp$^{2}$', fontsize=18)
        plt.tight_layout()
        plt.grid()
        plt.grid('on','minor')
        if save_png == True:
            plt.savefig(basename+'_amp2_calibration.png')
        if show == True:
            plt.show()
        plt.clf()

        plt.plot(freq, np.sqrt(PSD[:,0]), linewidth=1)
        plt.tick_params(axis='both', labelsize=15)
        plt.title('\nPowerspectrum', fontsize=24)
        plt.suptitle('used: %s\nNFFT: %s'%(basename, NFFT), fontsize=9)
        plt.xlabel('Frequency [Hz]', fontsize=18)
        plt.ylabel('amp', fontsize=18)
        plt.tight_layout()
```

```
        plt.grid()
        plt.grid('on','minor')
        if save_png == True:
            plt.savefig(basename+'_amp_calibration.png')
        if show == True:
            plt.show()
        plt.clf()

        plt.plot(freq, 20*np.log10(np.sqrt(PSD[:,0])), linewidth=1)
        plt.tick_params(axis='both', labelsize=15)
        plt.title('\nPowerspectrum', fontsize=24)
        plt.suptitle('used: %s\nNFFT: %s'%(basename, NFFT), fontsize=9)
        plt.xlabel('Frequency [Hz]', fontsize=18)
        plt.ylabel('dB re amp', fontsize=18)
        plt.tight_layout()
        plt.grid()
        plt.grid('on','minor')
        if save_png == True:
            plt.savefig(basename+'_dBreamp_calibration.png')
        if show == True:
            plt.show()
        plt.clf()

        plt.plot(freq, 20*np.log10(np.sqrt(PSD[:,0])/(1e-6)), linewidth=1)
        plt.tick_params(axis='both', labelsize=15)
        plt.title('\nPowerspectrum', fontsize=24)
        plt.suptitle('used: %s\nNFFT: %s'%(basename, NFFT), fontsize=9)
        plt.xlabel('Frequency [Hz]', fontsize=18)
        plt.ylabel('dB re $\mu$amp', fontsize=18)
        plt.tight_layout()
        plt.grid()
        plt.grid('on','minor')
        if save_png == True:
            plt.savefig(basename+'_dBremuamp_calibration.png')
        if show == True:
            plt.show()
        plt.clf()

    basename = basename + '_spectrum'

    return freq, PSD[:,0], basename

#================================================================================
# GET RATIO ELECTRONICS                                                           <==
#================================================================================

def Elec_ratio(wav_file, cal_freq, NFFT, lowcut, highcut):

    basename = os.path.splitext(os.path.basename(wav_file))[0]
    print('- Electronics of the piezo is being examined using:\n   %s' %basename)

    # get properties of the file
    data                = wav_file
    wavo                = wavio.read(data)  # returns data.shape, dtype, rate
                                            # and sampwitdt (sampwidth == 3
                                            # <-> 24 bits WAV)
    data                = wavo.data[:,0]    # this can be saved as a .dat file
    Fs                  = wavo.rate
    length              = wavo.data.shape[0]
    NFFT                = NFFT
```

```python
    # filter
    data = butter_bandpass_filter(data, basename, lowcut, highcut, Fs, order=5)[0]

    # Make - PSD - amp^2
    freq, PSD, basename = Spectrum(Fs, NFFT, data, basename)

    # Normalize
    sqrt_PSD = np.sqrt(PSD)
    PSD10kHz = sqrt_PSD[NFFT*cal_freq/Fs]     # check: wat is de waarde bij cal_freq?
    ratios = sqrt_PSD/PSD10kHz
    basename = basename + '_normalized'

    # save
    if save_txt == True:
        np.savetxt('20170419_6_white_noise_0.9Vrms_RATIO_nomalized.dat', ratios)

    # plot ratios
    if plotjes == True:
        plt.plot(freq, ratios, linewidth=1)
        plt.tick_params(axis='both', labelsize=15)
        plt.title('\n\n\nNormalized ratio electronics', fontsize=24)
        plt.suptitle('Freq at calibration freq %skHz equals one' %int(cal_freq/1000.
        plt.xlabel('Frequency [Hz]', fontsize=18)
        plt.ylabel('amp', fontsize=18) # only /$\sqrt(Hz)$ if PDS so NFFT = Fs
        plt.ylim(0, 10)
        plt.grid()
        plt.grid('on','minor')
        plt.tight_layout()
        if save_png == True:
            plt.savefig(basename+'_amp_calibration.png')
        if show == True:
            plt.show()
        plt.clf()

    # plot ratios on dB scale
    if plotjes == True:
        plt.plot(freq, 20*np.log10(ratios), linewidth=1)
        plt.tick_params(axis='both', labelsize=15)
        plt.title('\n\n\nNormalized ratio electronics')
        plt.suptitle('Freq at calibration freq %skHz equals one' %int(cal_freq/1000.
        plt.xlabel('Frequency [Hz]', fontsize=18)
        plt.ylabel('dB re amp',fontsize=18) # only sqrt(Hz) if PDS so NFFT = Fs
        plt.ylim(-20, 20)
        plt.grid()
        plt.grid('on','minor')
        plt.tight_layout()
        if save_png == True:
            plt.savefig(basename+'_dBreamp_calibration.png')
        if show == True:
            plt.show()
        plt.clf()

    return freq, ratios

#==============================================================================
# CORRECTION FOR DISTANCE
#==============================================================================

def distance_correction(config):
```

```python
    print('- Determine distance correction\n'
          '   based on 1 over distance squared relation')

    if config == 1:
        RD = 0.52        #[m]
        SD = 0.97        #[m]
        SR = SD - RD     #[m]
        print 'distance_correction has not been determined for this config'
    elif config == 2:
        RD = 0.52        #[m]
        SD = 0.97        #[m]
        SR = SD - RD     #[m]
    elif config == 3:
        RD = 0.52        #[m]
        SD = 0.97        #[m]
        SR = SD - RD     #[m]
        print 'distance_correction has not been determined for this config'
    elif config == 4:
        RD = 0.52        #[m]
        SD = 0.94        #[m]
        SR = SD - RD     #[m]
        print 'distance_correction has not been determined for this config'
    elif config == 5:
        RD = 0.52        #[m]
        SD = 0.94        #[m]
        SR = SD - RD     #[m]
        print 'distance_correction has not been determined for this config'
    elif config == 6:
        RD = 0.5         #[m]
        SD = 1.          #[m]
        SR = SD - RD     #[m]
        print 'distance_correction has not been determined for this config'
    elif congif ==7:
        RD = 1.5         #[m]
        SD = 2.          #[m]
        SR = SD - RD     #[m]

    distance_correction = SR**2/SD**2

    return distance_correction

#==============================================================================
# CALIBRATION VALUE
#==============================================================================

def IJk_value(hyd_data, basename, Fs, NFFT, FcumBegin, FcumEnd):

    Fs = Fs

    if Fs == int(25e6/128):
        print '- Determine calibration value of the DOM using:\n   %s' %basename
    elif Fs == 2e5:
        print '- Determine calibration value of the REF using:\n   %s' %basename

    # PSD in amp^2
    freq, PSD, basename = Spectrum(Fs, NFFT, hyd_data, basename)

    # Because noise measurement only shows peaks of electronics
    # we do not subtract noise.
```

```
    # determine cumsum
    dF        = float(Fs)/float(NFFT)    # amount of freq per bin
    idxB      = int(FcumBegin/dF)        # bin to start cumsum
    idxE      = int(FcumEnd/dF)          # bin to stop cumsum
    cummulative = PSD[idxB:idxE].cumsum();
    std = 10*np.log10(cummulative[-1]); #According to Parseval
    sqrt_std = np.sqrt(cummulative[-1])

    print '   sigma = %6.4f dB re amp, or sigma = %6.4f amp' %(std, sqrt_std)

    if plotjes == True:

        plt.plot(freq, np.sqrt(abs(PSD)), zorder = 1, linewidth=1) # PSD in dB re am
        #plot cummulatives
        plt.plot(freq[idxB:idxE], abs(cummulative), linewidth=1)
        plt.plot(freq[idxE:idxB-1:-1], abs(PSD[idxE:idxB-1:-1].cumsum()), linewidth=
        plt.tick_params(axis='both', labelsize=15)
        str = "$\sigma$ = %6.4f amp" % (sqrt_std)
        plt.title('\n\nPowerspectrum', fontsize=24)
        plt.suptitle('used: %s\nNFFT: %s\n %s' %(basename, NFFT, str), fontsize=9)
        plt.ylabel("amp ", fontsize=18)
        plt.autoscale(tight = True)
        plt.grid()
        plt.grid('on', 'minor')
        plt.xlabel('Frequency [Hz]', fontsize=18)
        plt.tight_layout()
        if save_png == True:
            plt.savefig(basename+'.png')
        if show == True:
            plt.show()
        plt.clf()

        plt.plot(freq, 10.*np.log10(abs(PSD)), zorder = 1, linewidth=1) # PSD in dB
        #plot cummulatives
        plt.plot(freq[idxB:idxE],10.*np.log10(abs(cummulative)), linewidth=1)
        plt.plot(freq[idxE:idxB-1:-1], 10.*np.log10(abs(PSD[idxE:idxB-1:-1].cumsum()
        plt.tick_params(axis='both', labelsize=15)
        str = "$\sigma$ = %6.4f dB re amp" % (std)
        plt.title('\n\nPowerspectrum', fontsize=24)
        plt.suptitle('used: %s\nNFFT: %s\n %s' %(basename, NFFT, str), fontsize=9)
        plt.ylabel("dB re amp", fontsize=18)
        plt.autoscale(tight = True)
        plt.grid()
        plt.grid('on', 'minor')
        plt.xlabel('Frequency [Hz]', fontsize=18)
        plt.tight_layout()
        if save_png == True:
            plt.savefig(basename+'_dB.png')
        if show == True:
            plt.show()
        plt.clf()

    calibration_value = sqrt_std

    return calibration_value

#==============================================================================
# PREPARE THE FILES USED FOR CALIBRATIONVALUES
#==============================================================================
```

```python
def get_cut_filt_IJkfile(file_ijkHYD, Fs, lowcut, highcut, NFFT):

    print '- Cut and filter data of:\n   %s' %file_ijkHYD

    Fs = Fs
    ijkHYD = np.genfromtxt(file_ijkHYD)
    basename = os.path.splitext(os.path.basename(file_ijkHYD))[0]
    ijkHYD, basename = butter_bandpass_filter(ijkHYD, basename, lowcut, highcut, Fs,
    ijkHYD, basename = Cut(ijkHYD, NFFT, Fs, basename)

    return ijkHYD, basename

#==============================================================================
# CALIBRATE
#==============================================================================

def CalibrationArray(file_electronics, file_ijkDOM, file_ijkREF, lowcut, highcut, ca
#                      .wav file       .dat file    .dat file     [Hz]     [Hz]
    print('\nGOING TO MAKE CALIBRATION ARRAY')

    elec_freq, elec_ratio = Elec_ratio(file_electronics, cal_freq, NFFT, lowcut, hig
#   dimension less, since it's a ratio

    ijkDOM, basenameDOM = get_cut_filt_IJkfile(file_ijkDOM, Fs_DOM, lowcut, highcut,
    ijkREF, basenameREF = get_cut_filt_IJkfile(file_ijkREF, Fs_REF, lowcut, highcut,

    DOM_val_X = IJk_value(ijkDOM, basenameDOM, int(25e6/128), NFFT, cal_min, cal_max
    REF_val_Pa = IJk_value(ijkREF, basenameREF, int(2e5), NFFT, cal_min, cal_max) #

    sensitivity_value = (DOM_val_X / REF_val_Pa)   # in [X]/[Pa]
    dist_cor = distance_correction(config = 2)
    calibration_value = sensitivity_value / dist_cor

    print '- Used values used for calibration:'
    print '   Sensitivity value (DOM/REF) = %s' %sensitivity_value
    print '   Distance correction value   = %s' %dist_cor
    print '   Resulting calibration value = %s' %calibration_value

    return elec_freq, elec_ratio, calibration_value

#==============================================================================
# MAIN
#==============================================================================

def main():
    #start timer
    startTime = datetime.now()

    # input values for calubration:
    cal_type         = 'wav10kHz'                                           # Sp
    file_electronics= '20170419_6_white_noise_0.9Vrms.WAV'         # .wav      # el
    file_ijkDOM      = '20170310_config2_DOM_6_v1_wavelet_10kHz.dat'  # .dat      # fi
    file_ijkREF      = 'Time_Conf2Domwavelet10kHzv1.dat'             # .dat      # fi
    lowcut           = 1000                                          # [Hz]      # lc
    highcut          = 80000                                         # [Hz]      # up
    cal_freq         = 10000                                         # [Hz]      # we
    samples_to_cut   = 1024                                          # [samp]    # sc
    NFFT             = samples_to_cut # do NOT change this!          # [samp]    # sc
    Fs_DOM           = int(25e6/128)                                 # [samp/s] # sc
```

```
    Fs_REF          = int(2e5)                                      # [samp/s] # sc
    cal_min         = 3500                                          # [Hz]     # lc
    cal_max         = 22500                                         # [Hz]     # up

    #get electronics ratio, and calibration value
    elec_freq, elec_ratio, calibration_value = CalibrationArray(file_electronics, fi
    #due to different Fs, elec_freq is slightly different from freq --> levert 1.6%

    print '\nNOW CALIBRATE THE DATA:'
    for fn in os.listdir('.'):

        #DOM
        if ".dat" in fn:
            Fs = Fs_DOM
        #REF
#        if "Time_Conf2Domwavelet10kHzv1.dat" in fn:
#            Fs = Fs_REF
        #SOURCE
#        if "Wavelet_10kHz_5Vpp.dat" in fn:                                     #sam
#            Fs = 1e6

            # get file --> Less datapoints for tone and sweep
            print '- %s' %fn
            basename = os.path.splitext(os.path.basename(fn))[0]
            if 'toon' in fn:
                data = np.genfromtxt(fn, max_rows = Fs/2)
            if 'sweep' in fn:
                data = np.genfromtxt(fn, max_rows = Fs/2)
            else:
                data = np.genfromtxt(fn)

            # filter file. Cutting possible, but not needed
            data, basename = butter_bandpass_filter(data, basename, lowcut, highcut,
            #data, basename = Cut(data, NFFT, Fs, basename)

            #CALIBRATE DATA FILE (timetrace)
            data = data / calibration_value
            basename = 'c_%s_' %(cal_type) + basename
            print '  save: %s.dat'%basename

            # save file
            if save_txt == True:
                np.savetxt(basename+'.dat', data)

            try:
                plot(data, basename, Fs, NFFT)
            except OverflowError:
                print 'OverflowError - Allocated too many blocks'
                print 'Skip plot of timetrace'

            PSD, basename = MakeSpectrum(Fs, NFFT, data, basename)              #<==

            #CALIBRATE DATA FILE (timetrace)
            PSD_ELcor = (np.sqrt(PSD) / elec_ratio)**2

            # save file
            if save_txt == True:
                np.savetxt(basename+'PSD.dat', PSD)
                np.savetxt(basename+'PSD_ELcor.dat', PSD_ELcor)
            print '  save: %sPSD.dat'%basename
```

```
        print '  save: %sPSD_ELcor.dat'%basename

        try:
            PlotSpectrum(Fs, NFFT, PSD, basename)
        except OverflowError:
            print 'OverflowError - Allocated too many blocks'
            print 'Skip plot of spectrum'

        try:
            PlotSpectrum(Fs, NFFT, PSD_ELcor, basename+'_ELcor')
        except OverflowError:
            print 'OverflowError - Allocated too many blocks'
            print 'Skip plot of spectrum'

    #end timer
    print datetime.now() - startTime

if __name__ == "__main__":
    sys.exit(main())
```