Time Series Analysis in Astrophysics
Compulsory Exercise 2

Nicholas Emborg Jannsen
(201205962)

August 8, 2016

1 Introduction

The aim of this compulsory exercise is to expand on the developed software from exercise 1. The routine should now also handle data sets that is non-coherent in time, allow the use of statistical weights, calculate the window function (`window`), develop a software which can locate frequencies and subtract them from the time series (`clean`) (but also return the these peaks), and perform a data filtering in a low-pass, high-pass, and band-pass filter (`filters`). All routines is again developed in the computational language Python.

2 Programs

2.1 Test and plot program: `test2` and `plot2`

To more easily use the developed software in this compulsory exercise I have created a routine named `test2` where all programs are run through a main function. From this main function sub-functions can be called and executed. Moreover, a routine for plotting is also made named `plot2`. These routines will not be presented here but the details/the code can be found in the appendix.

2.2 Power spectrum: `power2`

Not to make a review of the already developed software from exercise 1, I here go through the changes made to the program `power`. In the function call

```
def power2(data, f_interval=None, f_resolution=None, sampling=None,
           w_column=None)
```

the parameters `sampling` and `w_column` is added. The first is an optional sampling of data, and the last is to specify which column in `data` the weights are placed. Hence `data` is now a matrix containing the weights (`w`) and if `w_column` is not specified the program will automatically assume that the weights are placed in the 3. column.

To simplify the code I make a few handy definitions

```
# Handy definitions:
dt_min = min(np.diff(t))          # Minimum time interval
dt_max = max(np.diff(t))          # Maximum time interval
dt_med = np.median(np.diff(t))    # Median time interval
f_Nq = 1./(2*dt_min)              # Cyclic Nyquist frequency
f_md = 1./(2*dt_med)              # Cyclic median frequency
```

I will allow the time series to consist of 2 coincident times and non-coherent sampled data, however, a time ordering of the data is still needed. This is done by

```
# Checking if data is ordered in time:
if dt_min<0:
    sys.exit('Error: Data not ordered in time')
```

Compared to the 1. computational exercise, the structure of the how the routine checks whether `f_interval` and `f_resolution` is defined is a bit different

```
# If f_interval and f_resolution is not defined:
if f_interval==None and f_resolution==None:
    if dt_max>2*dt_med or math.isnan(f_Nq):
        f_interval = [0, f_md]
        f_resolution = 1./(t[-1]-t[0])
    else:
        f_interval = [0, f_Nq]
        f_resolution = 1./(t[-1]-t[0])
# If only f_resolution is defined:
elif f_interval==None and f_resolution!=None
                      and isinstance(f_resolution, float):
    f_resolution = f_resolution
    if dt_max>2*dt_med or math.isnan(f_Nq):
       f_interval = [0, f_md]
    else:
       f_interval = [0, f_Nq]
# If only f_interval is defined:
elif f_resolution==None and len(f_interval)==2:
    f_interval   = f_interval
    f_resolution = 1./(t[-1]-t[0])
# If both f_interval and f_resolution is defined:
elif f_interval!=None and f_resolution!=None:
    f_interval   = f_interval
    f_resolution = f_resolution
else:
    sys.exit('Error: Wrong input argument for "f_interval"')
```

The overall structure is the same (the routine still check the 4 possibilities if `f_interval` and `f_resolution` is/is not given), however, because the data is allowed to contain coincident times (here `f_Nq = Inf`) and unevenly sampled data, the Nyquist frequency is not valid in those cases. A "approximately Nyquist frequency", given by 1 over 2 times the median time step (`f_md = 1./(2*dt_med)`), is valid when the maximum time step is larger than 2 times the median time

step (`dt_max > 2*dt_med`).

As mentioned the software is made such the data can be sampled (though it is attended for oversampling (>1)). This is possible with a simple `elif` statement

```
# The frequency interval is found:
if sampling==None:
    f = np.arange(f_interval[0], f_interval[1], f_resolution)
elif sampling!=None:
    f = np.arange(f_interval[0], f_interval[1], f_resolution*sampling)
```

When weights are present in `data` the software now take them into account. Moreover, the routine also returns `alpha` and `beta`. This can be seen from the following code block

```
if np.size(data[0,:])>2:
    print "Calculating WITH weights"
    if w_column==None:
        w = data[:,2]
    elif w_column!=None:
        w = data[:,w_column]
    for i in range(len(f)):
        sinsum = np.sin(f[i]*2*np.pi*t)
        cossum = np.cos(f[i]*2*np.pi*t)
        s[i]  = sum(w*S*sinsum)
        c[i]  = sum(w*S*cossum)
        ss[i] = sum(w*sinsum**2)
        cc[i] = sum(w*cossum**2)
        sc[i] = sum(w*sinsum*cossum)
    # Calculate alpha, beta, P, A
    alpha = (s*cc-c*sc)/(ss*cc-sc**2)
    beta  = (c*ss-s*sc)/(ss*cc-sc**2)
    P     = alpha**2+beta**2
    A     = np.sqrt(P)
```

Testing software: `power2`

To test if the software `power2` takes the weights into account when calculating the power spectrum, we have here created a vector of random weights with values between 5-7. The signal is created such it contains 3 clear sinusoidal peaks all with an amplitude of 1 and with a frequency of 20 c/d, 50 c/d, 80 c/d, respectively. As can be seen in Fig. 1 the random weights result in different peak amplitudes for the 3 signals. Similar results were seen when running the routine a couple of times, hence, it work for its purpose.

2.3 Window function: `window`

The fact that time series are time-sampled rather than continuous means that the Fourier transform entails spectral artifacts into the Fourier spectrum. In other words the data sampling creates a so-called spectral *window function* in the Fourier spectrum. As the window function effectively is the quality limit of the data, from an astronomic point of view the appearance of the window function is first of all determined by the quality of the observations and, secondly, how

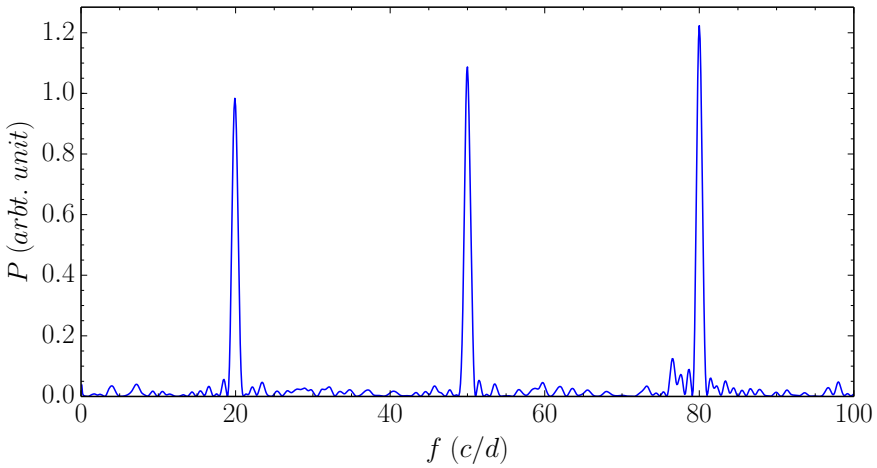

Figure 1: Power spectrum for a time varying signal constructed of 3 sinusoidal functions all with an amplitude of 1 and with frequencies of 20, 50, and 80 c/d, respectively. Here random weights between 5-7 is taking into account, which change the amplitudes of each peak.

the observations are sampled. In cases where the data is unevenly sampled or when a systematic signal either contained in the data itself or in the way it is sampled (e.g. the day-to-night sampling for ground based observations), the Fourier spectrum will additionally produce spectral artifacts.

The window function is constructed such it returns a vector of frequencies and power. It takes 5 arguments, where `data` is the only parameter which is not optional as seen here

```
def window(data, f_interval=None, f_resolution=None, sampling=None,
           w_column=None):
```

The function automatically finds the center of the specified frequency interval, and on the basic of this frequency and the given times, a sinusoidal and a co-sinusoidal function is calculated

```
f_range = round(f_interval[0] + (f_interval[1]-f_interval[0])/2)
fsin    = np.sin(2*np.pi*f_range*data0[:,0])
fcos    = np.cos(2*np.pi*f_range*data0[:,0])
```

Now the routine calculates the power spectrum by using the the sinusoidal and co-sinusoidal functions, the time data and the software `power2`

```
# Sinusoidal:
data[:,1] = fsin
Pf_power, _, = power2(data0, f_interval, f_resolution, sampling, w_column)
f    = Pf_power[:,0]
Psin = Pf_power[:,1]

# Co-sinusoidal:
data[:,1] = fcos
Pf_power, _, = power2(data0, f_interval, f_resolution, sampling, w_column)
f    = Pf_power[:,0]
Pcos = Pf_power[:,1]
```

As a final step the routine calculates the power spectrum for the window function by

```
P = 1./2*(Pcos+Psin)
```

Testing software: `window`:

I have tested the window function, again with a random sinusoidal signal of frequency 52 c/d, a random amplitude, and with random noise added. Fig. 2 display the power spectrum of the simulated time series (left) and the corresponding window function (right). I have used a sampling of 1, and the sum of the window function is 0.999410276 – this is in agreement with the role of thumb that the sum of the window function is approximate equal to the resolution. Again the routine was run a couple of times to make sure the window function changed when random noise and peaks amplitude also changed.

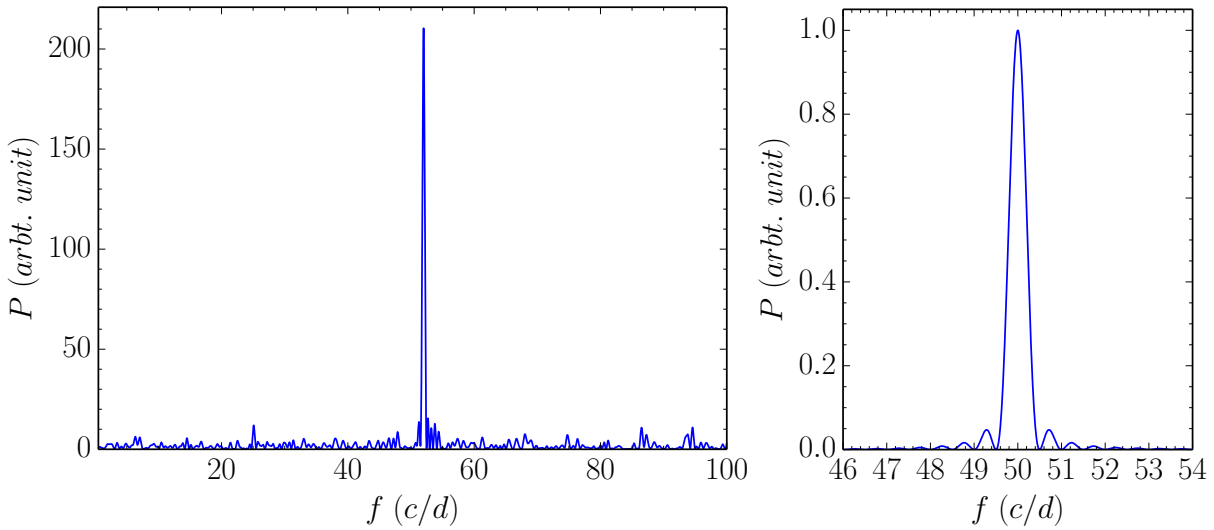

Figure 2: Left: Power spectrum for a time varying signal constructed of 1 sinusoidal function with a random amplitude and frequency (f = 52 c/d). Random noise is also added to the data and the data is not weighted. Right: Window function centered around 50 c/d for the data seen in the left-hand plot.

2.4 Cleaning routine: `clean`

The function called `clean` identify and select a specified number of highest valued peaks. The peaks are determined with a high accuracy and is then subtracted from the times series. As a output the routine returns a "cleaned" time series (`St_clean`), a time series corresponding for the number of specified highest peaks, and the frequencies (`f_peaks`) and amplitudes (`A_peaks`) of the subtracted peaks found by the routine. In the following function `N_peaks` is the number of peaks that should be subtracted

```
def clean(data, N_peaks, f_interval=None, f_resolution=None, sampling=None):
```

Each peak is found by a total of 3 iterations, where the first looks like

```
for i in range(N_peaks):
    # 1. Iteration: start
    Pf_power, P_comp = power2(data0, f_interval, f_resolution, sampling,
                              w_column)
    f     = Pf_power[:,0];   P = Pf_power[:,1];  PP = np.array(P)
    P_max = np.nanmax(P);    j = np.where(PP==P_max)[0]
    f_int = [f[j-1], f[j+1]]
```

First, the power spectrum of the given data set is calculated by using `power2`. To accurately locate each frequency peak of maximum power, we start by using a specified frequency resolution. When the peak is located, a new frequency interval around this peak is made (`f_int`) and a new frequency resolution (`f_RES`) is used in the 2. and 3. iteration. From the lectures generally a good frequency resolution to use is $1/(10\Delta t)$, where Δt is the time span of the data. Thus, we use this resolution. Also in the 2. and 3. iteration the power spectrum is not oversampled. With a high resolution, α, β, and the located frequencies and amplitudes of the peaks are found

```
    alpha0 = alpha[j]*np.sin(2*np.pi*f[j]*data0[:,0])
    beta0  = beta[j] *np.cos(2*np.pi*f[j]*data0[:,0])
    data0[:,1] = data0[:,1]-alpha0-beta0
    alpha[i]   = alpha0[j]
    beta[i]    = beta0[j]
    f_peaks[i] = f[j]
    A_peaks[i] = np.sqrt(P[j])
```

As can be seen, `alpha0` and `beta0` (describing the peak) is subtracted from the data, and is set equal to the original copied data. By this procedure, next time the for-loop runs this peak will not be contained in `data0`, and the routine looks for the next highest peak. The loop continues until it has run for `N_peaks` number of times. Lastly, the software return a cleaned time series (`St_clean`) and the located peaks (`f_peaks` and `A_peaks`).

Testing software: `clean`

To test the software `clean` I have made a small routine that simulate N sinusoidal signals with random peak frequencies (between 1–100 c/d) and amplitudes (between 5–20). To see the effect of cleaning a time series, random noise is added to the combined signal. As can be seen in Fig. 3 only $N = 3$ was considered. The upper plot shows the power spectrum, and the lower panel the time series. The functionality of `clean` can seen by the fact that the 3 simulated peaks in the raw data (—) is subtracted in the cleaned data (—). To make sure that the right peaks are subtracted, the software was run several time.

2.5 Bandpass filters: `low_band_pass` and `high_pass`

A final routine `filters` was developed to specify the frequency boundaries for which the power spectrum should be analyzed. In the following the filter routine `filters` will be explained, which can be used to filter data in a band, low, and high pass. Defining the interval containing filtered data by $P_{\text{filter}}(i,j) = \alpha_i \sin(2\pi f_i t_j) + \beta_i \cos(2\pi f_i t_j)$ the band, low, and high pass filter is given

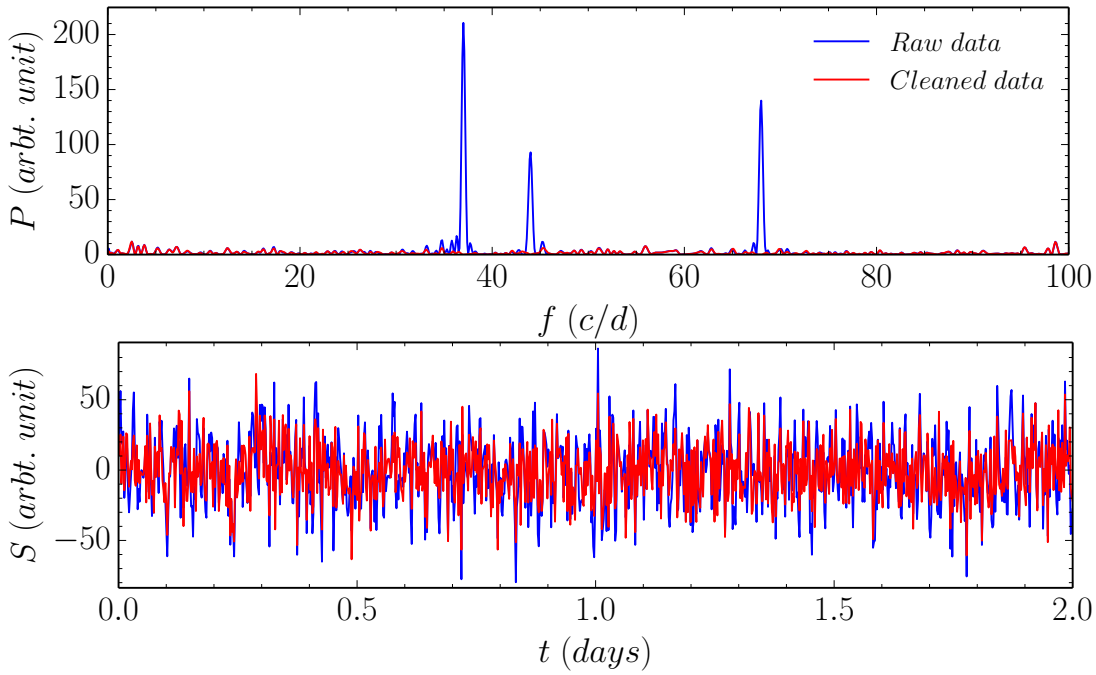

Figure 3: Upper: Power spectrum of 3 randomly simulated peaks with random noise added. Here the routine `clean` is used to distinguish between noise ((—)) and peak signals (—). Lower: Time series of data from upper panel. No oversampling and a resolution of 0.1 c/d.

by

$$S_{\text{band}} = \frac{\sum_{i=n(f_1)}^{n(f_2)} P_{\text{filter}}(i,j)}{\sum_{i=1}^{N} P_{\text{window}}(i)} \tag{1}$$

$$S_{\text{low}} = \frac{\sum_{i=1}^{n(f_{\text{low}})} P_{\text{filter}}(i,j)}{\sum_{i=1}^{N} P_{\text{window}}(i)} \tag{2}$$

$$S_{\text{high}} = \text{data}(j) - \frac{\sum_{i=1}^{n(f_{\text{high}})} P_{\text{filter}}(i,j)}{\sum_{i=1}^{N} P_{\text{window}}(i)} \tag{3}$$

The software takes the following arguments, where `f_interval` obviously is not an optional parameter

```
def filters(data, f_interval, f_resolution=None, sampling=None,
            w_column=None):
```

The routine can easily be used as a band, high, or low pass filter – this is purely determined by the frequency interval given. For convenience the software returns both data filtered in the wanted frequency interval (band or low pass), but also a high pass filter defined by the maximum limit of `f_interval` up to highest frequency available in the power spectrum. As seen from (3) the high pass filter is simply the remaining after the low pass filter subtracted from the data. From the definition given by (1), (2), and (3) the parameter $P_{\text{filter}}(i,j)$ is calculated by the following code

```
# Calculates P_filter:
P_filter = np.zeros(len(t))
for i in range(len(t)):
    alpha_sin    = alpha*np.sin(2*np.pi*f*t[i])
    beta_cos     = beta* np.cos(2*np.pi*f*t[i])
    P_filter[i] = sum(alpha_sin + beta_cos)
```

Here `alpha` and `beta` is constructed by the software `power2`. Now the window function, calculated by the software `window`, is used to filter out the given frequency interval from the time series

```
# Bandpass/Lowpass and Highpass filter:
S_low_band  = P_filter/sum(P_window)
S_high      = data[:,1]-S_low_band
```

Testing software: `filters`

A test of the bandpass filter was done by simulating 3 frequencies of 20, 50, and 80 c/d with amplitudes of 6, 10, and 7, respectively. To calculate the power spectrum I used again a time span of 2 days, a frequency resolution of 0.1 c/d, and no oversampling.

As seen in upper left panel of Fig. 4 a bandpass around the peak of maximum power was performed (a zoom-in on this peak can be seen in the upper right panel), and in the lower panel

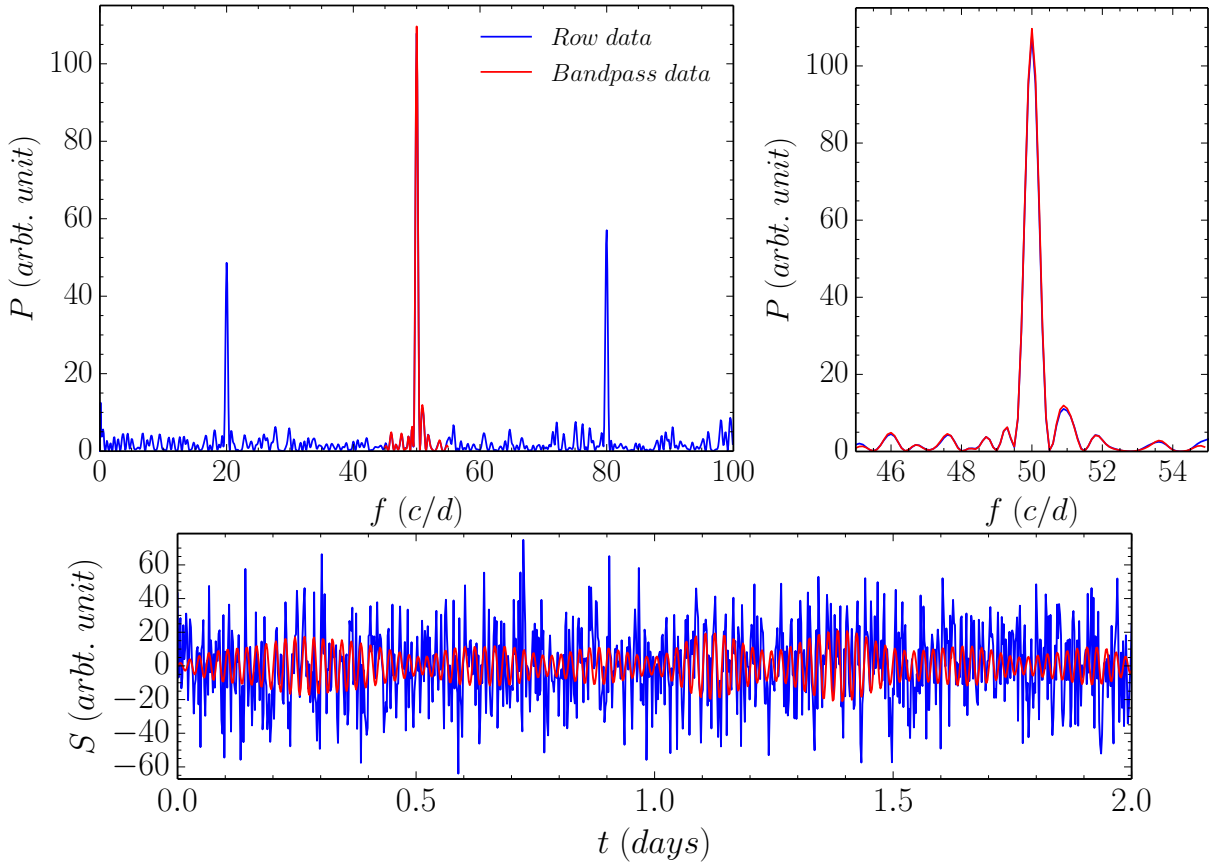

Figure 4: Left: Power spectrum of raw simulated data (—) and a bandpass centered around the peak maximum power (—). Right: A zoom-in on the bandpass region of the power spectrum. Bottom: Time series of raw (blue) and bandpass (red) data.

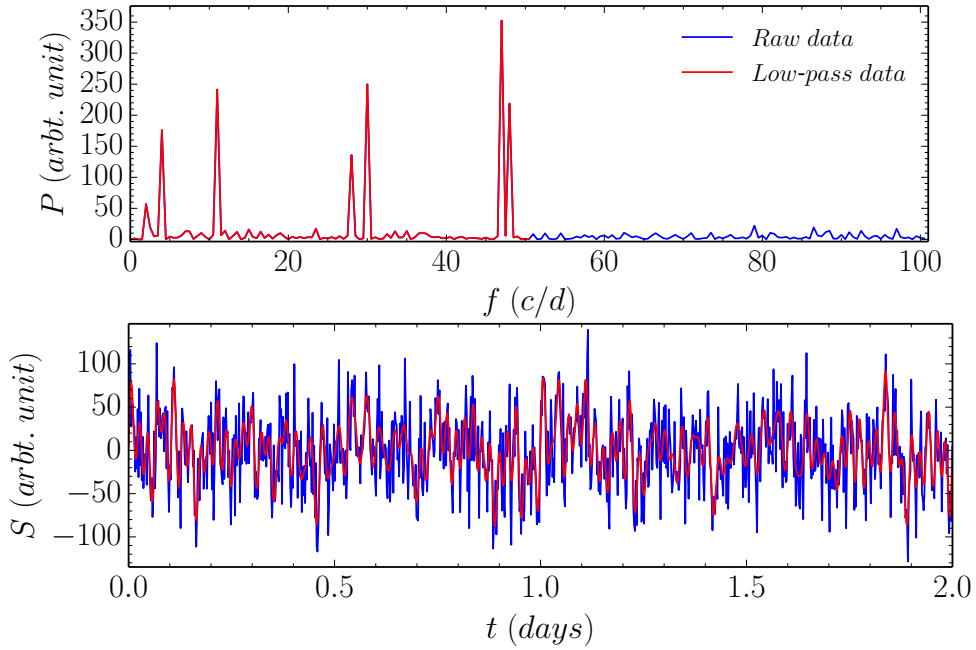

Figure 5: Upper: Power spectrum of raw simulated data (—) and a low-pass filter (—). Lower: Time series for the power spectra in upper panel.

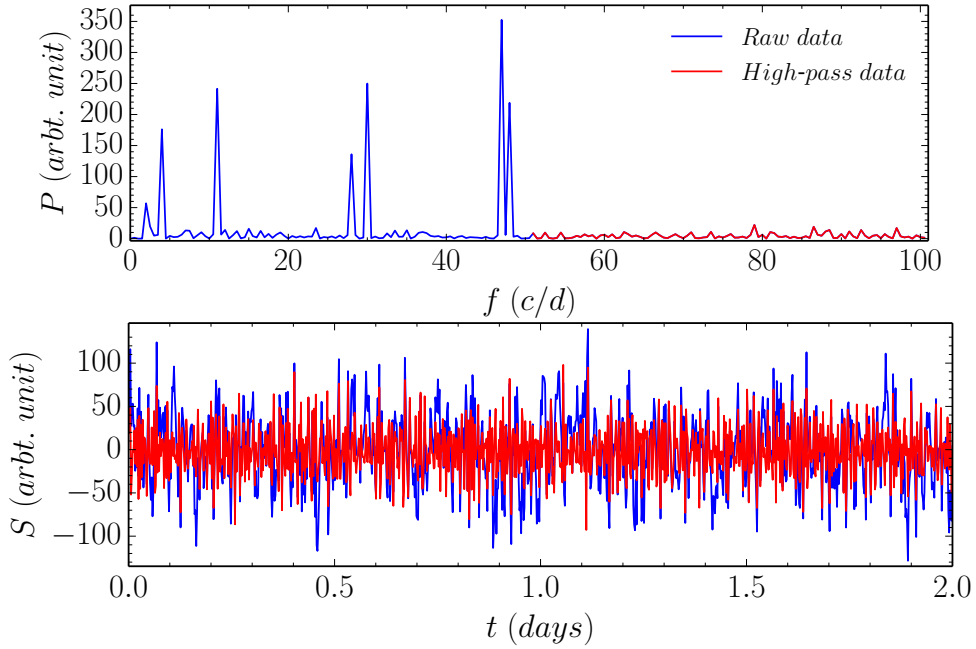

Figure 6: Upper: Power spectrum of raw simulated data (—) and a high-pass filter (—). Lower: Time series for the power spectra in upper panel.

the time series of the raw (—) and bandpass (—) data can be seen. The latter plot clearly illustrates the functionality of this filter. Same results were obtained when peak frequencies and amplitudes were changed.

Next a test of the low-pass and high-pass filter was made simultaneously, due to fact that the software `filters` allows this. To clearly see a difference I have simulated 7 peaks in the frequency interval between 0–50 c/d and added random noise over an interval spanning 0–100 c/d. Setting the low-pass interval to 0–50 c/d, the low-pass contains all significant peaks plus random noise, and the high-pass filter contain only random noise. The results can be seen in Fig. 5 and Fig. 6,

where the upper panel is the power spectra and the lower the time series. It is clear from the time series that the low-pass and high-pass do filter the desired frequency regions. The software was run several times for consistency.

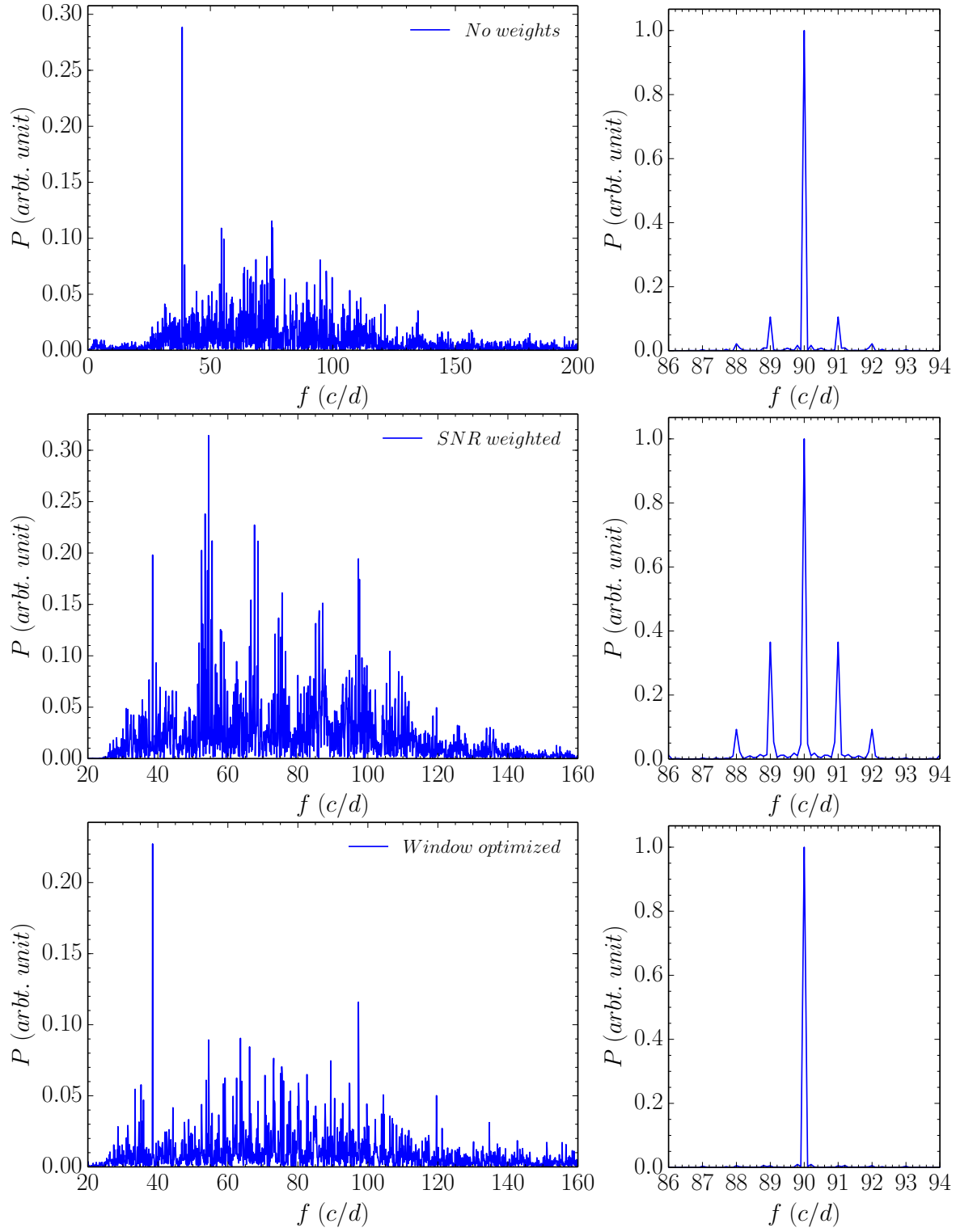

Figure 7: Left panels: Power spectrum of the star Procyon. Right panels: the window-function. The upper plots are no-weighted, middle plots are SNR-weighted (statistical weighted), and lower plots are Window-optimized (statistical weighted). All plots are with no oversampling and a resolution of 0.1 c/d.

3 Data Analysis

3.1 Exercise 1: Star Procyon

In this exercise the software `power2` is tested on data from the star Procyon. As seen in Fig. 7 the routine is run with no weights (upper panels), SNR weights (mid panels), and window optimized weights (lower panels). Here a frequency resolution of 0.1 c/d was used, and no oversampling. It is evident both from the power spectra and the window functions, that weighting have a clear effect on the data. From the window functions is can be seen that the biggest improvement of the quality spectrum is obtained with window optimized weights, whereas, the lowest quality spectrum is obtained when using SNR weights. Thus, because the SNR is too high it is not a good idea to use these as weights and the high side-lobes from the window function (panel mid right) can almost directly be seen in the power spectrum (panel mid left). Hence, the side-lobes directly contributes to the power which also is in good agreement with the fact that he SNR weighted power spectrum have higher peaks than the power spectra with no weighting and window optimized weighting.

3.2 Exercise 2: Kepler δ Scuti Star

In the analysis of a δ Scuti star observed with the Kepler spacecraft, the exercise is to clean the power spectrum. This is done with the software `clean` for the 7 most doninant peaks and the result be seen in Fig. 8. The upper panel display the raw (—) and cleaned (—) power spectrum, and the lower panel is the corresponding time series. The plotted frequency area is a zoom-in on the overall power spectrum, however, I insured that no other interesting features were visible from an inspection of the total spectrum.

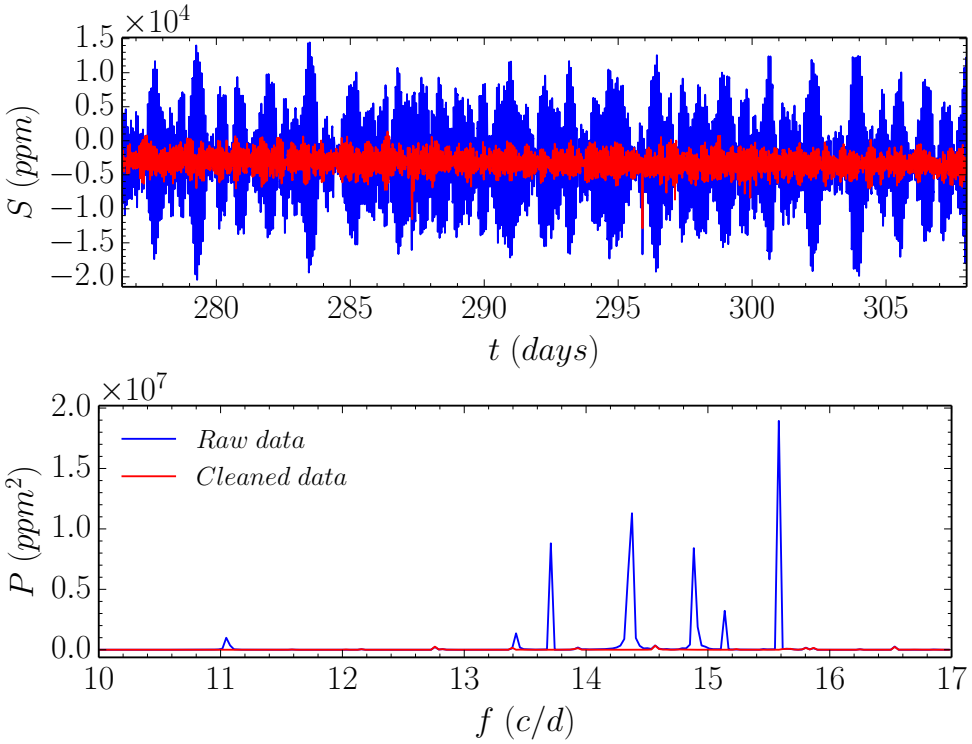

Figure 8: Upper: Power spectrum of raw (—) and cleaned (—) data for a δ Scuti star observed by Kepler. Lower: Time series for the power spectra in upper panel.

3.3 Exercise 3: Sun by SoHO/GOLF

Now the radial velocity data of the Sun observed by SoHO/GOLF instrument is ana. The exercise is to filter the data using a low-pass filter of frequencies 0.02 mHz (~ 1.7 c/d), 0.1 mHz (~ 8.6 c/d), and 1 mHz (~ 86.4 c/d). As the software `power2` uses frequencies in c/d we need first to convert mHz to c/d. The proper conversion is that $1\,\mathrm{mHz} = (24 \times 60 \times 60) \times 10^{-3}\,\mathrm{c/d} = 86.4\,\mathrm{c/d}$. The result of each filter can be seen in Fig. 9. From the upper panel and down, a low-pass filter up to 0.02 mHz, 0.1 mHz, and 1 mHz is visible, respectively. It can be seen that a higher low-pass frequency, introduce more power, and what is actually seen from (—) is mostly noise from granulation.

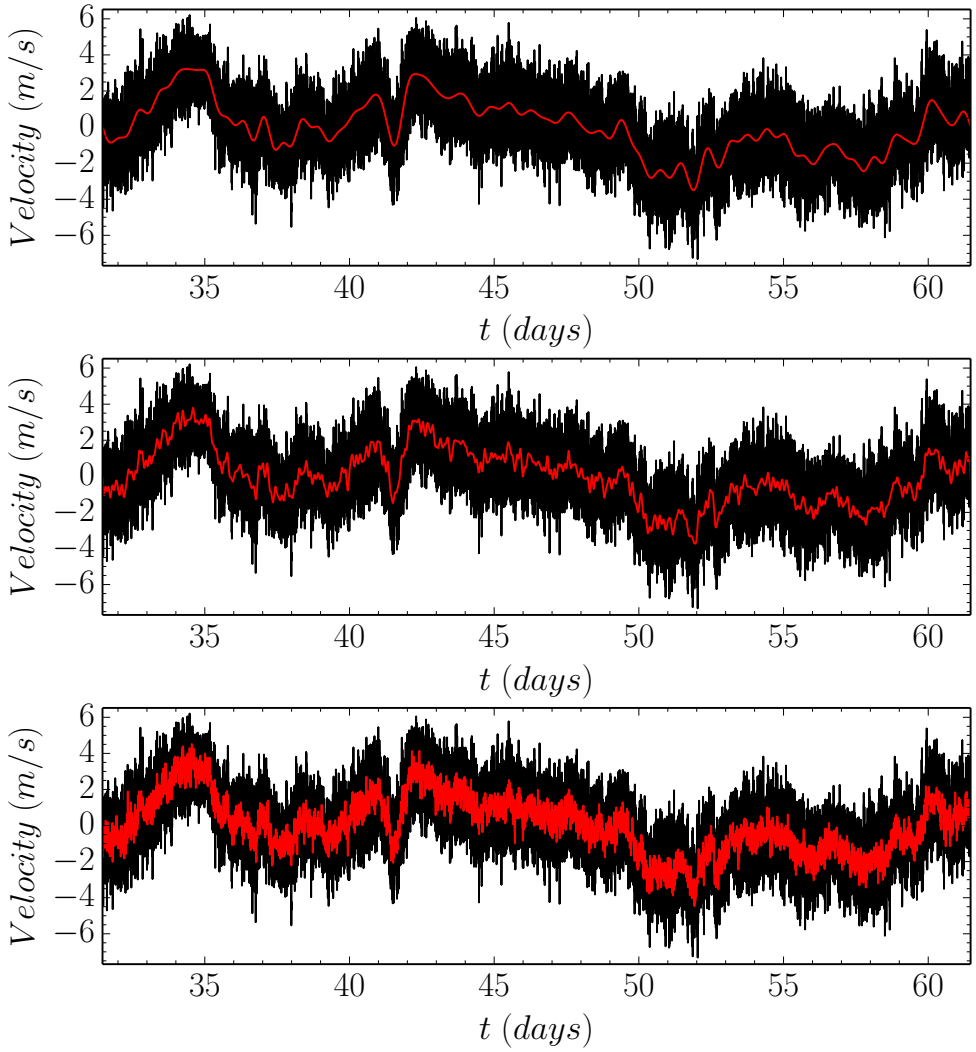

Figure 9: First panel: Time series of raw (—) and low-pass filter (—) for the Sun. Going from the upper panel and down a low-pass filter up to f =0.02 mHz, 0.1 mHz, and 1 mHz is used, respectively. Here no oversampling and a frequency resolution of `f_md` given by the data was used. The data is observed by SoHO/GOLF instrument.

3.4 Exercise 4: Sun by SoHO/GOLF

As seen in Fig. 10 a further analysis of the solar data was to select three oscillation modes with frequencies near 2 mHz (~ 173 c/d), 3 mHz (~ 259 c/d), and 4 mHz (~ 346 c/d) and make a narrow band-pass filter around these frequencies (see upper plots). In the upper panels of Fig. 10

the band-pass is spanning a peak at $f_2 = 162 - 164$ (—), $f_3 = 260 - 263$ (—), and $f_4 = 246 - 254$ (—). When comparing the time series for these 3 oscillations, one see that the life time of each mode becomes shorter with increasing frequency (here spanning from days for the low frequency peak to hours for the high frequency peak). Another thing to notice is that the peak(s) of maximum power is the placed at ~ 262 c/d, which nicely match with the fact that the Sun stochastically excites most p-modes in the range 200-350 c/d.

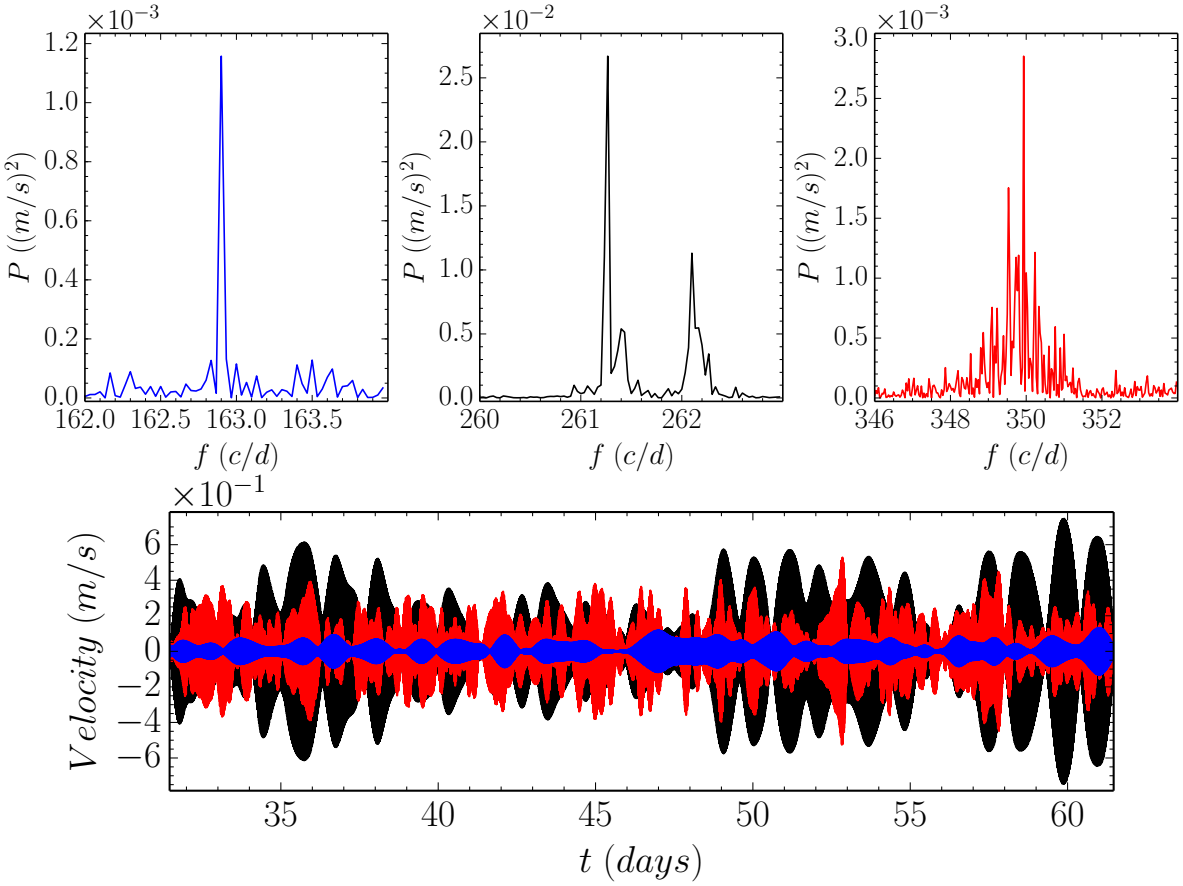

Figure 10: Upper panels: Power spectra for a band-pass close to a peak at 163 c/d (—), 262 c/d (modes of same radial order) (—), and 350 c/d (—). Lower panel: Time series for the band-pass filter seen in the upper panels.

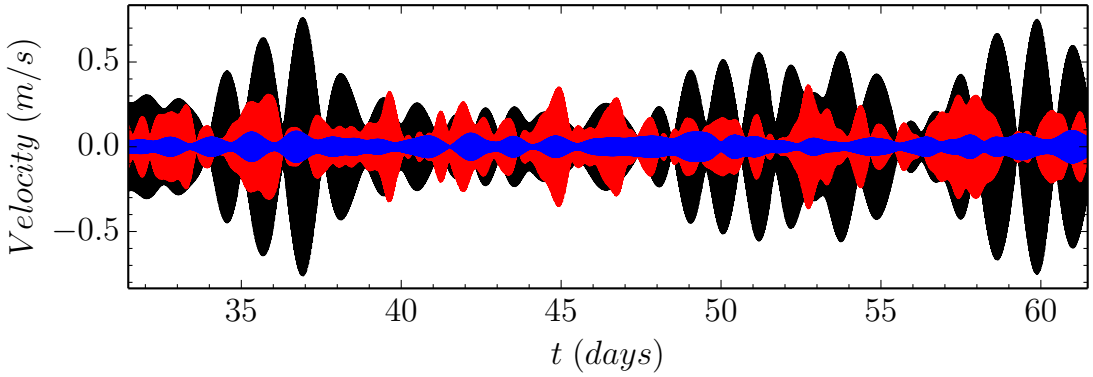

Figure 11: Time series for extracted peaks with the software `clean` close to a peak at 163 c/d (—), 262 c/d (modes of same radial order) (—), and 350 c/d (—). For the above mentioned 10, 20, and 30 peaks were extracted with `clean`.

3.5 Exercise 5: Sun by SoHO/GOLF

In this exercise we want to compare the power removal of the band-pass filter and the clean software. The justification of doing so, is that the `clean` software can be used to remove a large number of peaks, and thus the power spectrum returned from `clean` will resemble that of the band-pass – which was used in exercise 4 to remove the specified peaks. Now instead of removing the most dominating signal peaks, these were extracted from the time series such a comparission to the band-pass filters were possible. The extracted peak signals in the time series is shown in Fig. 11. A number of 10, 20, and 30 peaks were extracted from the 2 mHz, 3 mHz, and 4 mHz signal peak, respectively.

To make it more easy to compare the efficiency of the software `clean` and `filters` for these specific cases, I have subtracted the corresponding time series. The results can be seen in Fig. 12. Agian going from the upper plot and down, the time series for the 2 mHz, 3 mHz, and 4 mHz are shown. As one might expect one see that `clean` is dependent on the number of cleaned peaks, hence, less signal is deleted from the data compared to the band-pass filtered data. The cleaned power spectra was inspected to ensure that the peaks were extracted correctly.

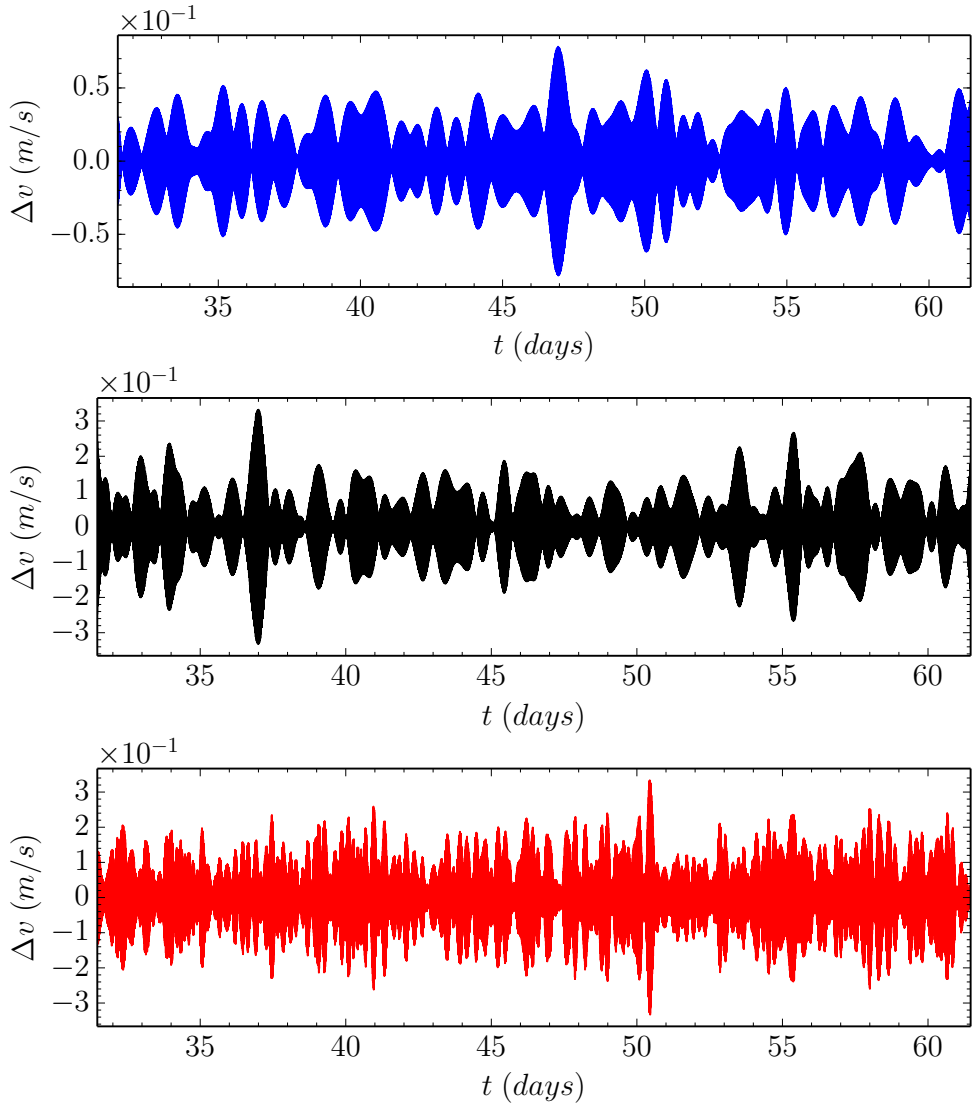

Figure 12: From upper to lower panel: Band-pass minus clean time series (Δv) for a peak close to 163 c/d (—), 262 c/d (—), and 350 c/d (—).

3.6 Exercise 6: Kepler Solar-like Stars

We now return to the data of 2 solar-like stars observed with Kepler. I have used my software to make a high-pass filter for these stars such no oscillations (what I know of) are present in the data. The frequency limit between the low-pass and high-pass filter was for both stars set to 400 c/d. The software was run with no oscillations and a frequency resolution set by the program to ~ 0.034 c/d. The resulting high-pass filter time series for star 1 (upper panel) and star 2 (lower panel) can be seen in Fig. 13.

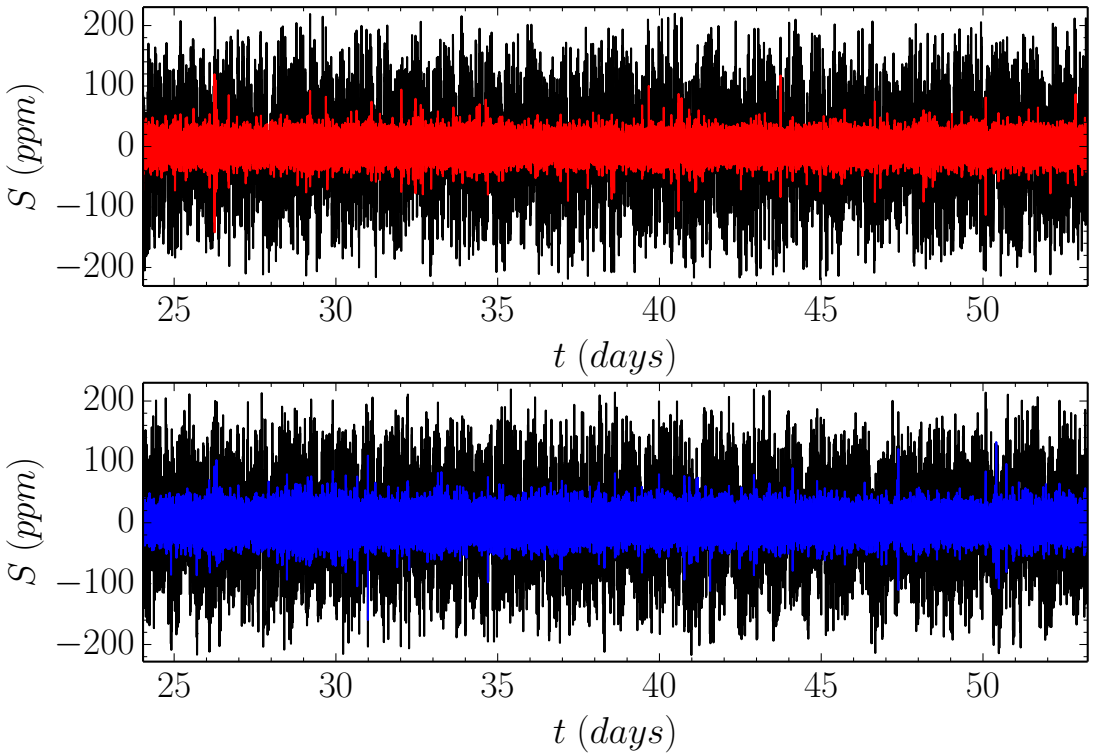

Figure 13: Upper panel: Raw (—) and high-pass filter (—) time series for solar-like star nr. 1. Lower panel: Raw (—) and high-pass filter (—) time series for the solar-like star nr. 2. The high-pass filter was set to a minimum frequency of 400 c/d for star 1 and 300 c/d for star 2. The data for both stars was observed with the Kepler telescope.

Next the scatter as a function of time for the high-pass filtered data can be estimated. This was done by using the following formula for the mean value (μ) and the scatter/standard deviation (σ)

$$\mu_i = \frac{1}{N+1} \sum_{j=i-N/2}^{i+N/2} \text{data}(t_j) \tag{4}$$

$$\sigma_i^2 = \frac{1}{N+1} \sum_{j=i-N/2}^{i+N/2} [\text{data}(t_j) - \mu_i]^2 \,. \tag{5}$$

Here N is an even number (and effectively the number of points use in the so-called *box car* filter). The best value of N depends on the sampling and on how fast the data quality is changing throughout the time series, however, a typical value is between 50–100.

To calculate σ_i I use a already developed function from the bottleneck library called `move_std`. As the name suggest, on the basis of the time series and a specified number data points that should be included in the calculation, this function calculates the scatter as a function of time.

Including 50 data points in the calculation, the the weights $w_i = \sigma_i^{-2}$ can be calculated. Fig. 14 display the result when comparing the power spectra for star 1 (red plots) and star 2 (blue plot) with the power spectra when including high frequency noise as statistical weights. It can be seen that there is only a small modification to the power spectra (most dominant for low frequencies), hence, the high frequency scatter for these solar-like stars are almost constant.

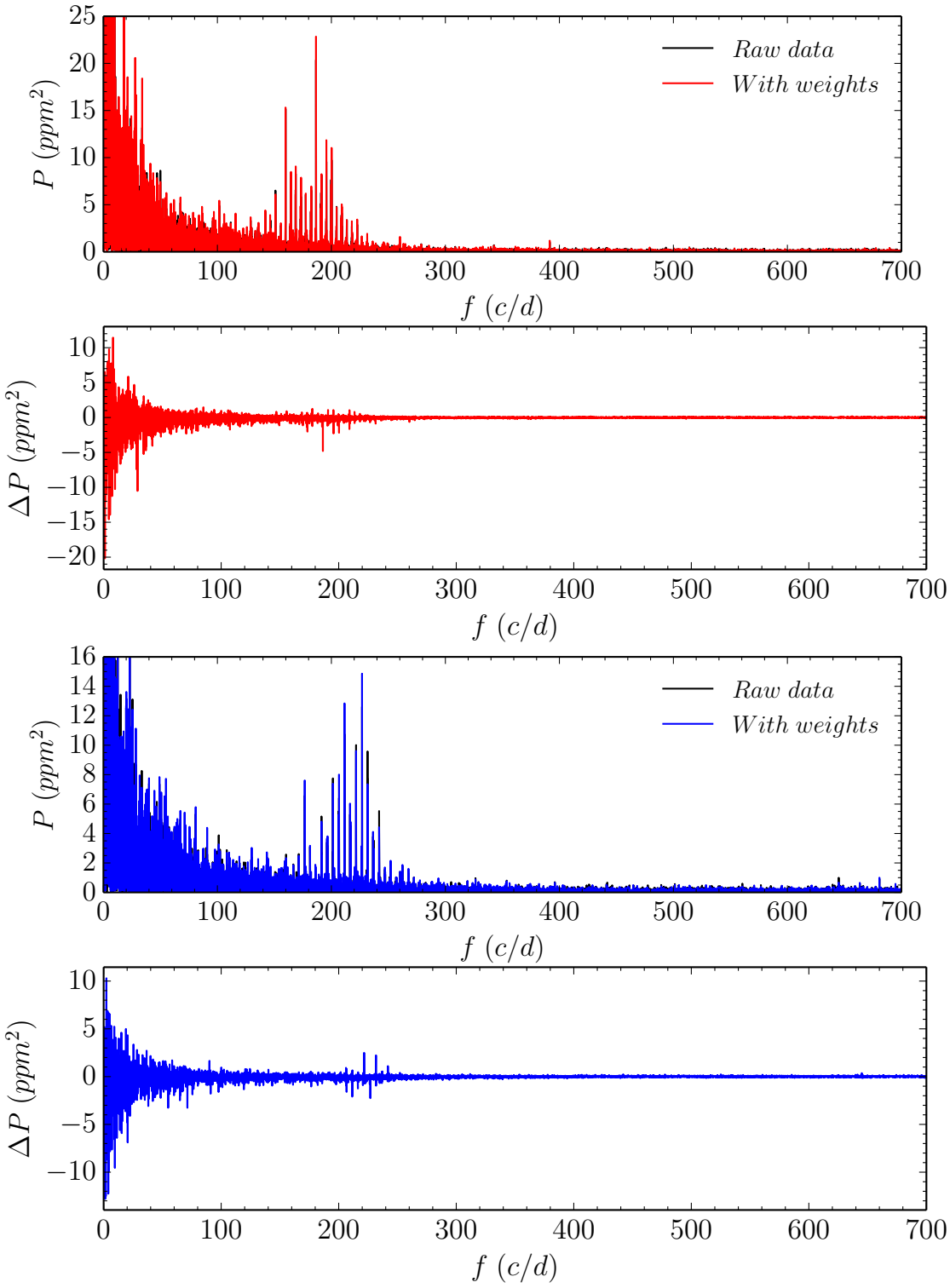

Figure 14: Two upper panels: The difference between power spectra from the raw times series for star 1 (—). Two lower panels: Same plot but here for star 2 (—). Panel 3 and 4 shows the direct difference ΔP for star 1 and 2, respectively.

Appendix – The Code

Software: power2

```
import math, sys, time
import numpy as np
import matplotlib.pyplot as plt
from numpy import inf

def power2(data, f_interval=None, f_resolution=None, sampling=None, w_column=None):
#---------------------------------------- FUNCTION ----------------------------------------:
# This function calculate the power spectrum of a time varying data set.
#----------INPUT:
# data           : Is a matrix with a column of time, signal, and possible weights.
# f_interval     : Frequency interval containing [fmin, fmax] (optional).
# f_resolution   : Frequency resolution (optional).
# sampling       : Sampling (oversampling >1) of the data (optional).
# w_column       : Column number where weights are placed in data (optional).
#                  w_column = 0 (calculates without weights)
#----------OUTPUT:
# Pf_data        : Frequencies (f) and power (P).
# P_comp         : Components for defining P (alpha and beta).
#-------------------------------------------------------------------------------------------:
    print "---------------------------------------- power2"
    start_time = time.time()  # Take time

    # Checking data structure:
    if ( np.size(data[0,:])<2   and                  # 2 columns minimum
         np.size(data[:,0])<2   and                  # 2 rows minimum
         isinstance(data,float)):                    # Text
         sys.exit('Error: Wrong data structure')     # Abort

    # Replacing nan by 0:
    if sum(sum(data))==np.nan:
        data[data==np.nan]=0
    # Replacing inf by 0:
    if sum(sum(data))==inf or sum(sum(data))==-inf:
        data[data== inf]=0
        data[data==-inf]=0

    # Splitting data:
    t = data[:,0]                                    # [days]
    S = data[:,1]-np.mean(np.array(data[:,1]))       # [arbt. unit]

    # Handy definitions:
    dt_min = min(np.diff(t))          # Minimum time interval
    dt_max = max(np.diff(t))          # Maximum time interval
    dt_med = np.median(np.diff(t))    # Median time interval
    f_Nq = 1./(2*dt_min)              # Cyclic Nyquist frequency
    f_md = 1./(2*dt_med)              # Cyclic median frequency

    # Checking if data is ordered in time:
    if dt_min<0:
        sys.exit('Error: Data not ordered in time')

    # Checking the existance of f_interval and f_resolution:
    # If f_interval and f_resolution is not defined:
    if f_interval==None and f_resolution==None:
        if dt_max>2*dt_med or math.isnan(f_Nq):
            print "Using f_md         = %f c/d" %f_md
```

```
        f_interval = [0, f_md]
        f_resolution = 1./(t[-1]-t[0])
    else:
        print "Using f_Nq          = %f c/d" %f_Nq
        f_interval = [0, f_Nq]
        f_resolution = 1./(t[-1]-t[0])
# If only f_resolution is defined:
elif f_interval==None and f_resolution!=None and isinstance(f_resolution, float):
    f_resolution = f_resolution
    if dt_max>2*dt_med or math.isnan(f_Nq):
        print "Using f_md          = %f c/d" %f_md
        f_interval = [0, f_md]
    else:
        print "Using f_Nq          = %f c/d" %f_Nq
        f_interval = [0, f_Nq]
# If only f_interval is defined:
elif f_resolution==None and len(f_interval)==2:
    f_interval   = f_interval
    f_resolution = 1./(t[-1]-t[0])
# If both f_interval and f_resolution is defined:
elif f_interval!=None and f_resolution!=None:
    f_interval   = f_interval
    f_resolution = f_resolution
else:
    sys.exit('Error: Wrong input argument for "f_interval"')
print "Using f_interval    = %s c/d" %f_interval
print "Using f_resolution  = %s c/d" %f_resolution

# The frequency interval is found:
if sampling==None:
    f = np.arange(f_interval[0], f_interval[1], f_resolution)
elif sampling!=None:
    print "Using sampling      = %f" %sampling
    f = np.arange(f_interval[0], f_interval[1], f_resolution*sampling)

# The power spectrum is calculated:
s  = np.zeros(len(f))
c  = np.zeros(len(f))
ss = np.zeros(len(f))
cc = np.zeros(len(f))
sc = np.zeros(len(f))
# Without weights
if w_column==0 or np.size(data[0,:])==2:
    print "Calculating WITHOUT weights"
    for i in range(len(f)):
        sinsum = np.sin(f[i]*2*np.pi*t)
        cossum = np.cos(f[i]*2*np.pi*t)
        s[i]  = sum(S*sinsum)
        c[i]  = sum(S*cossum)
        ss[i] = sum(sinsum**2)
        cc[i] = sum(cossum**2)
        sc[i] = sum(sinsum*cossum)
        if i==round(len(f)*0.25):
            print ("Done 25 procent --- %s seconds ---" % (time.time() - start_time))
        if i==round(len(f)*0.50):
            print ("Done 50 procent --- %s seconds ---" % (time.time() - start_time))
        if i==round(len(f)*0.75):
            print ("Done 75 procent --- %s seconds ---" % (time.time() - start_time))
if np.size(data[0,:])>2:
    print "Calculating WITH weights"
    if w_column==None:
        w = data[:,2]
```

```
        elif w_column!=None:
            w = data[:,w_column]
        for i in range(len(f)):
            sinsum = np.sin(f[i]*2*np.pi*t)
            cossum = np.cos(f[i]*2*np.pi*t)
            s[i]  = sum(w*S*sinsum)
            c[i]  = sum(w*S*cossum)
            ss[i] = sum(w*sinsum**2)
            cc[i] = sum(w*cossum**2)
            sc[i] = sum(w*sinsum*cossum)
            if i==round(len(f)*0.25):
                print ("Done 25 procent --- %s seconds ---" % (time.time() - start_time))
            if i==round(len(f)*0.50):
                print ("Done 50 procent --- %s seconds ---" % (time.time() - start_time))
            if i==round(len(f)*0.75):
                print ("Done 75 procent --- %s seconds ---" % (time.time() - start_time))
    # Calculate alpha, beta, P, A
    alpha = (s*cc-c*sc)/(ss*cc-sc**2)
    beta  = (c*ss-s*sc)/(ss*cc-sc**2)
    P     = alpha**2+beta**2
    A     = np.sqrt(P)

    Pf_data = np.vstack([f, P]).T
    P_comp  = np.vstack([alpha, beta]).T
    return Pf_data, P_comp
```

Software: window

```
import numpy as np
import math
import copy
from power2 import power2

def window(data, f_interval=None, f_resolution=None, sampling=None, w_column=None):
#---------------------------------------- FUNCTION ----------------------------------------:
# This function calculate the so-called window function. The function calls the routine 'power2'. It
    returns the power spectrum of the window function.
#---------INPUT:
# data          : Is a matrix with a column of time, signal, and possible weights.
# f_interval    : Frequency interval [fmin, fmax] (optional).
# f_resolution  : Frequency resolution (optional).
# oversampling  : Oversampling of the data (optional).
# w_column      : Column the weights are placed (optional).
#--------OUTPUT:
# Pf_window     : Frequency and power of the window function.
#-------------------------------------------------------------------------------------------:

    # Avoid overwritting data:
    data0 = copy.deepcopy(data)

    f_range = round(f_interval[0]+(f_interval[1]-f_interval[0])/2)
    fsin    = np.sin(2*np.pi*f_range*data0[:,0])
    fcos    = np.cos(2*np.pi*f_range*data0[:,0])

    # Sinusoidal
    data0[:,1] = fsin
    Pf_power, _, = power2(data0, f_interval, f_resolution, sampling, w_column)
    f    = Pf_power[:,0]
    Psin = Pf_power[:,1]

    # Co-sinusoidal
```

```
    data0[:,1] = fcos
    Pf_power, _, = power2(data0, f_interval, f_resolution, sampling, w_column)
    f    = Pf_power[:,0]
    Pcos = Pf_power[:,1]

    P = 1./2*(Pcos+Psin)
    Pf_window = np.vstack([f, P]).T
    return Pf_window
```

Software: clean

```
import math
import numpy as np
import sys
import copy
from power2 import power2

def clean(data, N_peaks, f_interval=None, f_resolution=None, sampling=None, w_column=None):
#------------------------------------------ FUNCTION ----------------------------------------:
# This function indentify and select a sepcified number of highest valued peaks. The peaks are
    determined with a high accuracy and is then subtracted from the times series. As a output the
    routine returns this more "clean" time series.
#----------INPUT:
# data           : Is a matrix with a column of time, signal, and possible weights.
# f_interval     : Frequency interval containing [fmin, fmax] (optional).
# f_resolution   : Frequency resolution (optional).
# sampling       : Sampling (oversampling >1) of the data (optional).
# w_column       : Column where the weights are placed (optional).
#----------OUTPUT:
# St_clean       : Times (t) and a cleaned Signal (S).
# P_comp         : Components of P is alpha and beta.
# f_peaks        : Highest frequency peak value for subtracted peaks.
#-------------------------------------------------------------------------------------------:
    # Avoid overwritting data:
    data0    = copy.deepcopy(data)

    # Constants:
    SAMPLING = 1
    f_RES    = 1./(10*(data0[-1,0]-data[0,0]))

    f_peaks = np.zeros(N_peaks)
    A_peaks = np.zeros(N_peaks)
    for i in range(N_peaks):
        # 1. Iteration: start
        Pf_power, P_comp = power2(data0, f_interval, f_resolution, sampling, w_column)
        f     = Pf_power[:,0];    P = Pf_power[:,1];   PP = np.array(P)
        P_max = np.nanmax(P);     j = np.where(PP==P_max)[0]
        f_int = [f[j-1], f[j+1]]
        # 2. Iteration: uses now f_res and so on..
        Pf_power, P_comp = power2(data0, f_int, f_RES, SAMPLING, w_column)
        f     = Pf_power[:,0];    P = Pf_power[:,1];    PP = np.array(P)
        P_max = np.nanmax(P);     j = np.where(PP==P_max)[0]
        f_int = [f[j-1], f[j+1]]
        # 3. Iteration: last
        Pf_power, P_comp = power2(data0, f_int, f_RES, SAMPLING, w_column)
        f     = Pf_power[:,0];    P    = Pf_power[:,1];   PP = np.array(P)
        P_max = np.nanmax(P);     j    = np.wheffre(PP==P_max)[0]
        alpha = P_comp[:,0];      beta = P_comp[:,1]
        alpha0 = alpha[j]*np.sin(2*np.pi*f[j]*data0[:,0])
        beta0  = beta[j]* np.cos(2*np.pi*f[j]*data0[:,0])
        data0[:,1] = data0[:,1]-alpha0-beta0
```

```
        f_peaks[i] = f[j]
        A_peaks[i] = np.sqrt(P[j])

    St_clean = data0
    return St_clean, f_peaks, A_peaks
```

Software: filters

```
import numpy as np
import math
import sys
import copy
import operator
from power2 import power2
from window import window

def filters(data, f_interval, f_resolution=None, sampling=None, w_column=None):
#---------------------------------------- FUNCTION ----------------------------------------:
# This function takes a data set and a frequency interval and calcuates the power spectrum within
    this interval. The software can be used as a low-pass, band-pass, or a high-pass filter, which i
     solely determined by the frequency interval.
#----------INPUT:
# data            : Is a matrix with a column of time, signal, and possible weights.
# f_interval      : Frequency interval containing [fmin, fmax].
# f_resolution    : Frequency resolution (optional).
# sampling        : Sampling (oversampling >1) of the data (optional).
# w_column        : Column the weights are placed (optional).
#----------OUTPUT:
# St_low_band     : Time series for low/band-pass filter.
# St_high         : Time series for high-pass filter.
#-------------------------------------------------------------------------------------------:

    # Avoid overwritting data:
    data0 = copy.deepcopy(data)

    # Calculates power spectrum:
    Pf_power, P_comp = power2(data0, f_interval, f_resolution, sampling, w_column)
    t     = data0[:,0]
    f     = Pf_power[:,0]
    alpha = P_comp[:,0]
    beta  = P_comp[:,1]

    # Calculates P_filter:
    P_filter = np.zeros(len(t))
    for i in range(len(t)):
        alpha_sin   = alpha*np.sin(2*np.pi*f*t[i])
        beta_cos    = beta*np.cos(2*np.pi*f*t[i])
        P_filter[i] = sum(alpha_sin + beta_cos)

    # Calculates window function:
    Pf_window = window(data0, f_interval, f_resolution, sampling)
    P_window  = Pf_window[:,1]

    # Bandpass/Lowpass and Highpass filter:
    S_low_band  = P_filter/sum(P_window)
    S_high      = data0[:,1]-S_low_band
    St_low_band = np.vstack([t, S_low_band]).T
    St_high     = np.vstack([t, S_high]).T
    return St_low_band, St_high
```

Software: test2

```
import math, sys, time, os
import numpy as np
import random as rd
import bottleneck as bn
import matplotlib.pyplot as plt

from numpy import inf

from power2 import power2
from window import window
from clean import clean
from filters import filters
from statsmodels.nonparametric.smoothers_lowess import lowess
from Plotting import Plotting
from plot2 import plot_timeseries, plot_power, plot_weights, plot_window, plot_clean, plot_filters
from plot2 import plot_timeseries21, plot_timeseries22, plot_timeseries3, plot_power21
from plot2 import plot_power22, plot_power3

def test2(name, f_interval=None, f_resolution=None, sampling=None, w_column=None):
#------------------------------------------ FUNCTION ----------------------------------------:
# This function calls a subfunction by name and execute it.
#------------INPUT:
# name              : Name of called function.
# f_interval        : Frequency interval containing [fmin, fmax] (optional).
# f_resolution      : Frequency resolution (optional).
# sampling          : Sampling (oversampling >1) of the data (optional).
#-------------------------------------------------------------------------------------------------:

    # Test of weights:
    if name=="weights":
        t_int = [0, 1]
        f_int = [0, 101]
        amp = [1, 1, 1];
        ft  = [20, 50, 80];
        t   = np.arange(t_int[0], t_int[1], 1e-3)
        S0  = amp[0]*np.sin(2*np.pi*ft[0]*t)
        S1  = amp[1]*np.cos(2*np.pi*ft[1]*t)
        S2  = amp[2]*np.sin(2*np.pi*ft[2]*t)
        S   = S0 + S1 + S2
        w   = np.random.normal(5,7,len(S))
        data = np.vstack([t, S, w]).T
        # Funtions:
        Pf_power, _, = power2(data, f_int, f_resolution, sampling)
        # Plotting:
        f = Pf_power[:,0]
        P = Pf_power[:,1]
        Plotting(f, P, xlab='$Frequency$ $[c/d]$', ylab='$Power$  $[arb. unit]$',\
                 xlim=f_int, ylim=None, mark='b-', legend='Weights')

    # Test of window:
    if name=="window":
        N = 1
        f_interval = [0.1, 100]
        t_interval = [0, 2]
        amp   = rd.sample(xrange(5,20),N)        # Random Aplitudes   (integer)
        ft    = rd.sample(xrange(1,100),N)       # Random Frequencies (integer)
        noise = np.random.normal(0,100,1e3)/5    # Random noise signal
        print "Random Frequencies: %s" %ft
        t     = np.linspace(t_interval[0], t_interval[1], 1e3)
        f     = [amp[i]*np.sin(2*np.pi*ft[i]*t) for i in range(N)]  # N frequency columns
```

```
    S    = np.sum(f,0)+noise                                    # add columns and add noise
    data = np.vstack([t, S]).T
    # Functions:
    Pf_power, _, = power2(data, f_interval, f_resolution, sampling)
    Pf_window    = window(data, f_interval, f_resolution, sampling)
    print sum(Pf_window)
    print("--- %s seconds ---" % (time.time() - start_time))
    # Plotting:
    plot_power( Pf_power,  f_interval)
    plot_window(Pf_window, f_interval)

# Test of clean:
if name=="clean":
    N = 20
    f_interval = [0, 100]
    t_interval = [0, 2]
    amp   = rd.sample(xrange(5,50),N)        # Random Aplitudes   (integer)
    ft    = rd.sample(xrange(1,100),N)       # Random Frequencies (integer)
    noise = np.random.normal(0,100,1000)/5   # Random noise signal
    print "Random Frequencies: %s" %ft
    t    = np.linspace(t_interval[0], t_interval[1], 1e3)
    f    = [amp[i]*np.sin(2*np.pi*ft[i]*t) for i in range(N)]  # N frequency columns
    S    = np.sum(f,0)+noise                                    # add columns and add noise
    data = np.vstack([t, S]).T
    # Functions:
    Pf_power, _, = power2(data,      f_interval, f_resolution, sampling, w_column)
    St_clean, _, = clean(data, N,    f_interval, f_resolution, sampling, w_column)
    Pf_clean, _, = power2(St_clean, f_interval, f_resolution, sampling, w_column)
    # Plotting:
    plot_clean(data, St_clean, t_interval, Pf_power, Pf_clean, f_interval)

# Test of filter:
if name=="filters":
    N = 7
    f_int0 = [0, 101]; f_int1 = [0, 51]; f_int2 = [51, 101]
    t_int0 = [0, 2]
    amp   = rd.sample(xrange(10,20),N)       # Random Aplitudes   (integer)
    ft    = rd.sample(xrange(1,100),N)        # Random Frequencies (integer)
    noise = np.random.normal(0,101,1e3)/3    # Random noise signal
    print "Random frequencies %s" %ft
    print "Random amplitudes  %s" %amp
    t     = np.linspace(t_int0[0], t_int0[1], 1e3)
    f     = [amp[i]*np.sin(2*np.pi*ft[i]*t) for i in range(len(ft))]
    S     = np.sum(f,0)+noise
    data  = np.vstack([t, S]).T
    # functions:
    Pf_power, _,     = power2(data,     f_int0, f_resolution, sampling, w_column)
    St_low, St_high = filters(data,    f_int1, f_resolution, sampling, w_column)
    Pf_low, _,       = power2(St_low,  f_int0, f_resolution, sampling, w_column)
    Pf_high, _,      = power2(St_high, f_int0, f_resolution, sampling, w_column)
    print("--- %s seconds ---" % (time.time() - start_time))
    # Plotting:
    plot_filters(data, St_low, St_high, t_int0, Pf_power, Pf_low, Pf_high, f_int0)

# Exercise 1 - Porcyon:
if name=="procyon":
    f_interval = [20, 160]
    data = np.loadtxt(os.path.join('/home/nicholas/Dropbox/Uni/Timeseries/Data', name))
    Pf_power, _,  = power2(data, f_interval, f_resolution, sampling, w_column=3)
    Pf_window     = window(data, f_interval, f_resolution, sampling, w_column=3)
    print("--- %s seconds ---" % (time.time() - start_time))
    plot_power (Pf_power,  f_interval)
```

```
    plot_window(Pf_window, f_interval)

# Exercise 2 - delta Scuti:
if name=="scuti":
    N = 1
    name = "kepler-star3-delta-scuti"
    data = np.loadtxt(os.path.join('/home/nicholas/Dropbox/Uni/Timeseries/Data', name))
    # Functions:
    Pf_power, _, = power2(data,      f_interval, f_resolution, sampling, w_column)
    St_clean, _, = clean(data, N,    f_interval, f_resolution, sampling, w_column)
    Pf_clean, _, = power2(St_clean, f_interval, f_resolution, sampling, w_column)
    print("--- %s seconds ---" % (time.time() - start_time))
    # Plotting:
    t_interval = [data[0,0], data[-1,0]];  f_interval = [0, 1./(data[-1,0]-data[0,0])]
    plot_clean(data, St_clean, t_interval, Pf_power, Pf_clean, f_interval)

# Sun SoHO/GOLF:
if name=="sun":
    name  = "solar-velocity"
    data  = np.loadtxt(os.path.join('/home/nicholas/Dropbox/Uni/Timeseries/Data', name))
    data[:,1] = data[:,1]-np.mean(np.array(data[:,1]))    # Correct for zero frequency
    t_int = [data[0,0], data[-1,0]]; f_int = [0, 555]
    peak0 = [162, 164];  N0 = 10
    peak1 = [260, 263];  N1 = 20
    peak2 = [346, 354];  N2 = 30
    # Exercise 3:
    # f_int = [0, 1*86.4]
    # Pf_power, _, = power2( data, f_int, f_resolution, sampling, w_column)
    # St_low, _,   = filters(data, f_int, f_resolution, sampling, w_column)
    # print("--- %s seconds ---" % (time.time() - start_time))
    # plot_timeseries2(data, St_low, t_int)
    # Exercise 4:
    # St_band0, _, = filters(data,     peak0, f_resolution, sampling, w_column)
    # St_band1, _, = filters(data,     peak1, f_resolution, sampling, w_column)
    # St_band2, _, = filters(data,     peak2, f_resolution, sampling, w_column)
    # Pf_band0, _, = power2(St_band0, peak0, f_resolution, sampling, w_column)
    # Pf_band1, _, = power2(St_band1, peak1, f_resolution, sampling, w_column)
    # Pf_band2, _, = power2(St_band2, peak2, f_resolution, sampling, w_column)
    # print("--- %s seconds ---" % (time.time() - start_time))
    # plot_timeseries3(St_band0, St_band1, St_band2, t_int)
    # plot_power(Pf_band0, peak0); plot_power(Pf_band1, peak1); plot_power(Pf_band2, peak2)
    # Exercise 5:
    #----- Peak 0:
    St_band0, _,      = filters(data,      peak0, f_resolution, sampling, w_column)
    St_clean0, _, _, = clean(data, N0,    peak0, f_resolution, sampling, w_column)
    # Pf_power0, _,     = power2(data,       peak0, f_resolution, sampling, w_column)
    # Pf_clean0, _,     = power2(St_clean0, peak0, f_resolution, sampling, w_column)
    S_rest0  = data[:,1]-St_clean0[:,1]
    St_rest0 = np.vstack([data[:,0], S_rest0]).T
    S_sub0   = St_band0[:,1]-S_rest0
    St_sub0  = np.vstack([data[:,0], S_sub0]).T
    #----- Peak 1:
    St_band1, _,      = filters(data,      peak1, f_resolution, sampling, w_column)
    St_clean1, _, _, = clean(data, N1,    peak1, f_resolution, sampling, w_column)
    # Pf_power1, _,     = power2(data,       peak1, f_resolution, sampling, w_column)
    # Pf_clean1, _,     = power2(St_clean1, peak1, f_resolution, sampling, w_column)
    S_rest1  = data[:,1]-St_clean1[:,1]
    St_rest1 = np.vstack([data[:,0], S_rest1]).T
    S_sub1   = St_band1[:,1]-S_rest1
    St_sub1  = np.vstack([data[:,0], S_sub1]).T
    #----- Peak 2:
    St_band2, _,      = filters(data,      peak2, f_resolution, sampling, w_column)
```

```
    St_clean2, _, _, = clean(data, N2,    peak2, f_resolution, sampling, w_column)
    # Pf_power2, _,     = power2(data,       peak2, f_resolution, sampling, w_column)
    # Pf_clean2, _,     = power2(St_clean2, peak2, f_resolution, sampling, w_column)
    S_rest2  = data[:,1] - St_clean2[:,1]
    St_rest2 = np.vstack([data[:,0], S_rest2]).T
    S_sub2   = St_band2[:,1] - S_rest2
    St_sub2  = np.vstack([data[:,0], S_sub2]).T
    # plotting:
    print("--- %s seconds ---" % (time.time() - start_time))
    plot_timeseries3(St_rest0, St_rest1, St_rest2, t_int)
    # plot_timeseries(St_sub0, t_int)
    # plot_power21(Pf_power0, Pf_clean0, peak1)
    # plot_timeseries(St_sub1, t_int)
    # plot_power21(Pf_power1, Pf_clean1, peak1)
    # plot_timeseries(St_sub2, t_int)
    # plot_power21(Pf_power2, Pf_clean2, peak2)

# Exercise 6 - Solar-like stars:
if name=="solar":
    name0  = "kepler-star1-solar-like"
    name1  = "kepler-star2-solar-like"
    data0  = np.loadtxt(os.path.join('/home/nicholas/Dropbox/Uni/Timeseries/Data', name0))
    data1  = np.loadtxt(os.path.join('/home/nicholas/Dropbox/Uni/Timeseries/Data', name1))
    t_int0 = [data0[0,0], data0[-1,0]];  f_int0 = [0, 400] # No peaks are visible above
    t_int1 = [data1[0,0], data1[-1,0]];  f_int1 = [0, 400] # No peaks are visible above
    f_int  = [0, 700]
    #-------- i) High-pass filtering:
    # Pf_power0, _, = power2(data0, f_int, f_resolution, sampling, w_column)
    # Pf_power1, _, = power2(data1, f_int, f_resolution, sampling, w_column)
    # np.savetxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star1_Pf2.txt', Pf_power0)
    # np.savetxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star2_Pf2.txt', Pf_power1)
    # _, St_high0   = filters(data0,   f_int0, f_resolution, sampling, w_column)
    # _, St_high1   = filters(data1,   f_int1, f_resolution, sampling, w_column)
    # Pf_high0, _,  = power2(St_high0, [400, 700],  f_resolution, sampling, w_column)
    # Pf_high1, _,  = power2(St_high1, [400, 700],  f_resolution, sampling, w_column)
    # print("--- %s seconds ---" % (time.time() - start_time))
    # plot_timeseries21(data0, St_high0, t_int0)
    # plot_timeseries22(data1, St_high1, t_int1)
    # plot_power21(Pf_power0,  Pf_high0, f_int)
    # plot_power22(Pf_power1,  Pf_high1, f_int)
    # # Save data:
    # # print len(data0[:,0]), len(St_high0[:,1]), len(Pf_high0[:,0]), len(Pf_high0[:,1])
    # # print len(data1[:,0]), len(St_high1[:,1]), len(Pf_high1[:,0]), len(Pf_high1[:,1])
    # star1_St = np.vstack([data0[:,0],     St_high0[:,1]]).T
    # star1_Pf = np.vstack([Pf_high0[:,0], Pf_high0[:,1]]).T
    # star2_St = np.vstack([data1[:,0],     St_high1[:,1]]).T
    # star2_Pf = np.vstack([Pf_high1[:,0], Pf_high1[:,1]]).T
    # np.savetxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star1_St.txt', star1_St)
    # np.savetxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star1_Pf.txt', star1_Pf)
    # np.savetxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star2_St.txt', star2_St)
    # np.savetxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star2_Pf.txt', star2_Pf)
    #-------- ii) Scattter:
    ## Load data:
    # St_star1 = np.loadtxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star1_St.txt')
    # St_star2 = np.loadtxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star2_St.txt')
    # std1 = bn.move_std(St_star1[:,1], window=50, min_count=1)
    # std2 = bn.move_std(St_star2[:,1], window=50, min_count=1)
    # w1   = std1**-2
    # w2   = std2**-2
    # Data1 = np.vstack([data0[:,0], data0[:,1], w1]).T
    # Data2 = np.vstack([data1[:,0], data1[:,1], w2]).T
    # Pf_power1, _, = power2(Data1, f_int, f_resolution, sampling)
```

```
        # Pf_power2, _, = power2(Data2, f_int, f_resolution, sampling)
        # np.savetxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star1_Pfw.txt', Pf_power1)
        # np.savetxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star2_Pfw.txt', Pf_power2)
        ## Plotting:
        Pf1  = np.loadtxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star1_Pf2.txt')
        Pf2  = np.loadtxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star2_Pf2.txt')
        Pfw1 = np.loadtxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star1_Pfw.txt')
        Pfw2 = np.loadtxt('/home/nicholas/Dropbox/Uni/Timeseries/Data/solar_star2_Pfw.txt')
        dP1  = Pf1[:,1] - Pfw1[:,1]
        dP2  = Pf2[:,1] - Pfw2[:,1]
        # Plot:
        plot_power22(Pf1, Pfw1, f_int)
        plot_power22(Pf2, Pfw2, f_int)
        plot_power(np.vstack([Pf1[:,0], dP1]).T, f_int)
        plot_power(np.vstack([Pf2[:,0], dP2]).T, f_int)
        print("--- %s seconds ---" % (time.time() - start_time))


    return

if __name__ == "__main__": #-------------------- Main function --------------------#
    # FUNCTION CALL:
    start_time = time.time()  # Take time
    test2(name="solar", f_interval=None, f_resolution=None, sampling=1, w_column=None)
```

Software: plot2

```
import matplotlib.pyplot as plt
import numpy as np
import time
from matplotlib import gridspec as gs
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset


# Settings ------------------------------------------------------------:
def plot_settings():
    FS = 16
    plt.rc('text', usetex=True)
    plt.rc('xtick', labelsize=FS)
    plt.rc('ytick', labelsize=FS)
    plt.xlabel('xlabel', fontsize=FS)
    plt.ylabel('ylabel', fontsize=FS)
    plt.minorticks_on()
    # plt.ticklabel_format(axis='y',style='sci', scilimits=(0,3))
    # plt.tight_layout()

def plot_subplot1():  # Normal
    g  = gs.GridSpec(3, 7)
    ax = plt.subplot(g[0:2, 0:5])

def plot_subplot2():  # window
    g  = gs.GridSpec(3, 7)
    ax = plt.subplot(g[0:2, 4:7])

def plot_subplot3():  # Thin and broad
    g  = gs.GridSpec(5, 1)
    ax = plt.subplot(g[0:2])

def plot_subplot4():  # Normal
    g  = gs.GridSpec(3, 7)
```

```
    ax = plt.subplot(g[0:2, 0:7])

def St_axes(t_min, t_max, S):
    wd = (np.nanmax(S)-min(S))*0.05
    plt.xlim(t_min, t_max)
    plt.ylim(min(S)-wd, np.nanmax(S)+wd)
    plt.xlabel('$t$ $(days)$')
    plt.ylabel('$Velocity$ $(m/s)$')

def Pf_axes(x_min, x_max, y):
    ws    = (np.nanmax(y) - np.nanmin(y))*0.05
    y_min = 0  #np.nanmin(y)-ws
    y_max = 25 #np.nanmax(y)+ws
    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)
    plt.xlabel('$f$ $(c/d)$')
    plt.ylabel('$P$ $(ppm^2)$')

#-----------------------------------------------------------------------:

def plot_timeseries(data, t_int):
    t = data[:,0];  S = data[:,1]
    t_min = t_int[0]; t_max = t_int[1]
    plot_subplot3()
    plot_settings()
    plt.plot(t, S, 'r-')
    St_axes(t_min, t_max, S)
    plt.show()

def plot_timeseries21(data0, data1, t_int):
    t0 = data0[:,0];  S0 = data0[:,1]
    t1 = data1[:,0];  S1 = data1[:,1]
    t_min = t_int[0]; t_max = t_int[1]
    plot_subplot3()
    plot_settings()
    plt.plot(t0, S0, 'k-')
    plt.plot(t1, S1, 'r-')
    St_axes(t_min, t_max, S0)
    plt.show()

def plot_timeseries22(data0, data1, t_int):
    t0 = data0[:,0];  S0 = data0[:,1]
    t1 = data1[:,0];  S1 = data1[:,1]
    t_min = t_int[0]; t_max = t_int[1]
    plot_subplot3()
    plot_settings()
    plt.plot(t0, S0, 'k-')
    plt.plot(t1, S1, 'b-')
    St_axes(t_min, t_max, S0)
    plt.show()


def plot_timeseries3(data0, data1, data2, t_int):
    t0 = data0[:,0];  S0 = data0[:,1]
    t1 = data1[:,0];  S1 = data1[:,1]
    t2 = data2[:,0];  S2 = data2[:,1]
    t_min = t_int[0]; t_max = t_int[1]
    plot_subplot3()
    plot_settings()
    plt.plot(t1, S1, 'k-')
    plt.plot(t2, S2, 'r-')
    plt.plot(t0, S0, 'b-')
```

```
    St_axes(t_min, t_max, S1)
    plt.show()

def plot_power(data, f_int):
    f  = data[:,0];   P  = data[:,1]
    f_min = f_int[0];   f_max = f_int[1]
    plot_subplot3()
    plot_settings()
    plt.plot(f, P, 'r-')
    # plt.legend(loc='upper right',numpoints=1,frameon=False,fontsize=14)
    Pf_axes(f_min, f_max, P)
    # plt.xticks(np.arange(f_min, f_max, 1))
    plt.show()

def plot_power21(data0, data1, f_int):
    f0 = data0[:,0];   P0 = data0[:,1]
    f1 = data1[:,0];   P1 = data1[:,1]
    f_min = f_int[0]; f_max = f_int[1]
    plot_subplot3()
    plot_settings()
    plt.plot(f0, P0, 'b-', label='$Raw$ $data$')
    plt.plot(f1, P1, 'm-', label='$High$-$pass$ $400$ $c/d$')
    plt.legend(loc='upper right',numpoints=1,frameon=False,fontsize=14)
    Pf_axes(f_min, f_max, P0)
    plt.show()

def plot_power22(data0, data1, f_int):
    f0 = data0[:,0];   P0 = data0[:,1]
    f1 = data1[:,0];   P1 = data1[:,1]
    f_min = f_int[0]; f_max = f_int[1]
    plot_subplot3()
    plot_settings()
    plt.plot(f0, P0, 'k-', label='$Raw$ $data$')
    plt.plot(f1, P1, 'r-', label='$With$ $weights$')
    plt.legend(loc='upper right',numpoints=1,frameon=False,fontsize=14)
    Pf_axes(f_min, f_max, P1)
    plt.show()

def plot_power3(data0, data1, data2, f_int):
    f0 = data0[:,0];   P0 = data0[:,1]
    f1 = data1[:,0];   P1 = data1[:,1]
    f2 = data2[:,0];   P2 = data2[:,1]
    f_min = f_int[0]; f_max = f_int[1]
    plot_subplot3()
    plot_settings()
    plt.plot(f0, P0, 'k-', label='$Raw$ $data$')
    plt.plot(f1, P1, 'r-', label='$High$-$pass$ $300$ $c/d$')
    plt.plot(f2, P2, 'b-', label='$hej$')
    plt.legend(loc='upper right',numpoints=1,frameon=False,fontsize=14)
    Pf_axes(f_min, f_max, P0)
    plt.show()


#----------------------------------------

def plot_weights(data0, data1, f_int):
    f  = data0[:,0];   P  = data0[:,1]
    fw = data1[:,0];   Pw = data1[:,1]
    f_min = f_int[0];   f_max = f_int[1]
    plot_settings()
    plt.plot(f, P)
    plt.plot(fw, Pw, 'r-')
```

```
    Pf_axes(f_min, f_max, Pw)
    plt.show()

def plot_window(data, f_interval):
    f  = data[:,0];  P  = data[:,1]
    f_min = f_interval[0]; f_max = f_interval[1]
    f_mid = round(f_min + (f_max-f_min)/2)
    plot_subplot2()
    plot_settings()
    plt.plot(f, P)
    Pf_axes(f_mid-4, f_mid+4, P)
    plt.show()

def plot_clean(St_data0, St_data1, t_interval, Pf_data0, Pf_data1, f_interval):
    # Time series:
    t  = St_data0[:,0];  S  = St_data0[:,1]
    tc = St_data1[:,0];  Sc = St_data1[:,1]
    t_min = t_interval[0]; t_max = t_interval[1]
    plot_subplot1()
    plot_settings()
    plt.plot(t, S)
    plt.plot(tc, Sc, 'r-')
    St_axes(t_min, t_max, S)
    plt.show()
    # Power spectrum:
    f  = Pf_data0[:,0];  P  = Pf_data0[:,1]
    fc = Pf_data1[:,0];  Pc = Pf_data1[:,1]
    f_min = f_interval[0]; f_max = f_interval[1]
    plot_subplot1()
    plot_settings()
    plt.plot(f, P, label='$Raw$ $data$')
    plt.plot(fc, Pc, 'r-', label='$Cleaned$ $data$')
    plt.legend(loc='upper left',numpoints=1,frameon=False,fontsize=14)
    Pf_axes(f_min, f_max, P)
    plt.show()

def plot_filters(St_data, St_low, St_high, t_int, Pf_data, Pf_low, Pf_high, f_int):
    # Time series:
    t0 = St_data[:,0];  S0 = St_data[:,1]
    t1 = St_low [:,0];  S1 = St_low [:,1]
    t2 = St_high[:,0];  S2 = St_high[:,1]
    t_min = t_int[0]; t_max = t_int[1]
    # Plot low:
    plot_subplot3()
    plot_settings()
    plt.plot(t0, S0, 'b-')
    plt.plot(t1, S1, 'r-')
    St_axes(t_min, t_max, S0)
    plt.show()
    # Plot high:
    plot_subplot3()
    plot_settings()
    plt.plot(t0, S0, 'b-')
    plt.plot(t2, S2, 'r-')
    St_axes(t_min, t_max, S0)
    plt.show()
    # Power spectrum:
    f0 = Pf_data[:,0];  P0 = Pf_data[:,1]
    f1 = Pf_low [:,0];  P1 = Pf_low [:,1]
    f2 = Pf_high[:,0];  P2 = Pf_high[:,1]
    f_min = f_int[0]; f_max = f_int[1]
    # Plot low:
```

```
plot_subplot3()
plot_settings()
plt.plot(f0, P0, 'b-', label='$Raw$ $data$')
plt.plot(f1, P1, 'r-', label='$Low$-$pass$ $data$')
plt.legend(loc='upper right', numpoints=1, frameon=False, fontsize=14)
Pf_axes(f_min, f_max, P0)
plt.show()
# Plot high:
plot_subplot3()
plot_settings()
plt.plot(f0, P0, 'b-', label='$Raw$ $data$')
plt.plot(f2, P2, 'r-', label='$High$-$pass$ $data$')
plt.legend(loc='upper right', numpoints=1, frameon=False, fontsize=14)
Pf_axes(f_min, f_max, P0)
plt.show()
# Power spectrum - Zoom in:
# f_min = 45; f_max = 55
# plot_subplot2()
# plot_settings()
# plt.plot(f, P)
# plt.plot(fc, Pc, 'r-')
# Pf_axes(f_min, f_max, P)
# plt.show()
```