Introduction

Contents

This is the work carried out for the second part of the Signal Processing SP exam at VKI 2022-23.

Exercise 1

Compute the velocity and acceleration by differentiating the signal.

For the differentiation, pick any finite difference formulation you consider appropriate.

Then, study the frequency content of the three signals (displacement, velocity and acceleration).

What do you see and why?

```
import numpy as np
from scipy import signal
import pandas as pd
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt, get_window, firwin, freqz
from scipy.signal.windows import general_hamming
%matplotlib inline
```

Finite difference differentiation

Functions

- **Diff_1st_order**

  Compute derivatives with first order backward difference approximation:

  $y'(x_i) = \frac{y(x_i) - y(x_{i-1})}{\Delta x}$

  $y''(x_i) = \frac{y(x_i) - 2y(x_{i-1}) + y(x_{i-2})}{\Delta x^2}$

  For the first point (firsts 2 for the 2nd order derivative), the forward difference approximation, that mirrors the expression above, is considered.

Note

In this way the first two elements will be just the opposite of each other. This problem could be addressed in different ways. Some of them:

1. Neglect the first derivative term. Since the firsts elements of the series don't measure important values.
2. Since, in fact, we have the datas for t<5s we could consider them to compute the backward derivative also at x=t=5s.

Note2

I could use simply: dy[0:-1] = np.diff(y)/np.diff(x) dy[-1] = (y[-1] - y[-2])/delta_x[-1] That would be equal to a forward first order diff approx, except at the last point where a backward approach is used.

- **Diff_2nd_order**

  Compute derivatives with second order central difference approximation:

  $y'(x_i) = \frac{y(x_{i+1}) - y(x_{i-1})}{2\Delta x}$

  $y''(x_i) = \frac{y(x_{i+1}) - 2y(x_i) + y(x_{i-1})}{\Delta x^2}$

  The derivatives for the 2 points at the edge, is computed with the forward differencing at the left and the backwards one on the right.

  Forward:

  $y'(x_i) = \frac{-y(x_{i+2}) + 4y(x_{i+1}) - 3y(x_i)}{2\Delta x}$

  $y''(x_i) = \frac{-y(x_{i+3}) + 4y(x_{i+2}) - 5y(x_{i+1}) + 2y(x_i)}{\Delta x^2}$

  Backward:

  $y'(x_i) = \frac{y(x_{i-2}) - 4y(x_{i-1}) + 3y(x_i)}{2\Delta x}$

  $y''(x_i) = \frac{-y(x_{i-3}) + 4y(x_{i-2}) - 5y(x_{i-1}) + 2y(x_i)}{\Delta x^2}$

Note

To notice: To compute the 2nd order difference schemes the pre-implemented python function **gradient** could be considered:
It uses second order accurate central differences in the interior points and second order accurate one-sides (forward or backwards) differences at the boundaries:
y_prime_2nd = np.gradient(y, x, edge_order=2)

My function and the gradient function were tested and compared leading to negligible differences regarding the first order derivative:
max(abs(error)) = 1.6122214674396673e-11

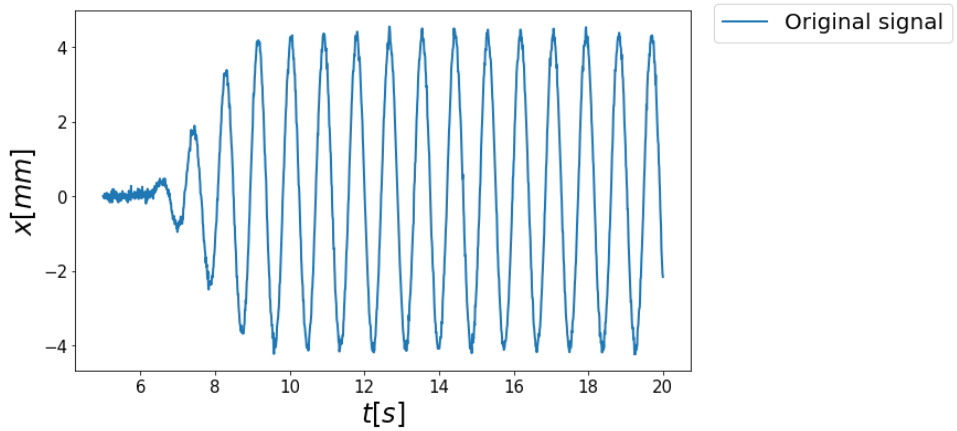
error = abs(y_prime_2nd - np.gradient(y, x, edge_order=2)) plt.plot(error) print(max(error))

```
def Diff_1st_order(x,y):
    """
    Compute derivatives with first order backward difference approximation:
    y'(xi) = (y(xi)−y(xi−1)) / Δx
    y''(xi) = (y(xi)−2y(xi−1)+y(xi-2) / Δx^2
    For the first point (firsts 2 for the 2nd order derivative),
    The forward difference approximation, that mirrors the expression above,
    is considered
    """

    N = x.shape[0]

    delta_x = np.empty(N)
    y_prime = np.empty(N) # 1st order derivative
    y_prime2 = np.empty(N) # 2nd order derivative
    # Forward first order differencing approx
    delta_x[0] = x[1] - x[0]
    y_prime[0] = (y[1] - y[0]) / delta_x[0]
    y_prime2[0] = (y[2] - 2*y[1] + y[0]) / delta_x[0]**2
    # Backward first order differencing approx
    for i in range(1, N):
        delta_x[i] = x[i] - x[i-1]
        y_prime[i] = (y[i] - y[i-1]) / delta_x[i]
        if i == 1:
            y_prime2[i] = (y[i+2] - 2*y[i+1] + y[i]) / delta_x[i]**2
        else:
            y_prime2[i] = (y[i] - 2*y[i-1] + y[i-2]) / delta_x[i]**2

    return y_prime, y_prime2

def Diff_2nd_order(x,y):
    """
    Compute derivatives with second order central difference approximation:
    y'(xi) = (y(xi+1)−y(xi−1)) / 2Δx
    y''(xi) = (y(xi+1)−2y(xi)+y(xi-1) / Δx^2
    The derivatives for the 2 points at the edge, is computed with the forward
    differencing at the left and the backwards one on the right.
    Forward:
    y'(xi) = (-y(xi+2)+4y(xi+1)-3y(xi)) / 2Δx
    y''(xi) = (-y(xi+3)+4y(xi+2)-5y(xi+1)+2y(xi)) / Δx^2
    Backward:
    y'(xi) = (y(xi-2)-4y(xi-1)+3y(xi)) / 2Δx
    y''(xi) = (-y(xi-3)+4y(xi-2)-5y(xi-1)+2y(xi)) / Δx^2
    """

    N = x.shape[0]

    delta_x = np.empty(N)
    # y_prime = np.gradient(y, x, edge_order=2) # 1st order derivative
    y_prime = np.empty(N) # 1st order derivative
    y_prime2 = np.empty(N) # 2nd order derivative

    # Forward first order differencing approx
    delta_x[0] = x[1] - x[0]
    y_prime[0] = (-y[2] + 4*y[1] - 3*y[0]) / (2*delta_x[0])
    y_prime2[0] = (-y[3] + 4*y[2] - 5*y[1] + 2*y[0]) / delta_x[0]**2
    # Central second order differencing approx
    for i in range(1, N):
        delta_x[i] = x[i] - x[i-1]
        if i == N-1:
            y_prime[i] = (3*y[i] - 4*y[i-1] + y[i-2]) / (2*delta_x[i])
            y_prime2[i] = (-y[i-3] + 4*y[i-2] - 5*y[i-1] + 2*y[i]) / delta_x[i]**2
        else:
            y_prime[i] = (y[i+1] - y[i-1]) / (2*delta_x[i])
            y_prime2[i] = (y[i+1] - 2*y[i] + y[i-1]) / delta_x[i]**2

    return y_prime, y_prime2
```

```
# Import data from Excel file

# We consider datas only from t=5s onwards
# For this reason we discard the first 500 values
columns = ['Time','ODS_raw']
n = 500
data_raw = pd.read_excel('Signal_raw092.xlsx',
                         header=None, skiprows=n+1, names=columns)
x, y = np.array(data_raw.Time), np.array(data_raw['ODS_raw'])

#print('Dataset starting from t=5s')
#print(data_raw)
```

Finite difference differentiation

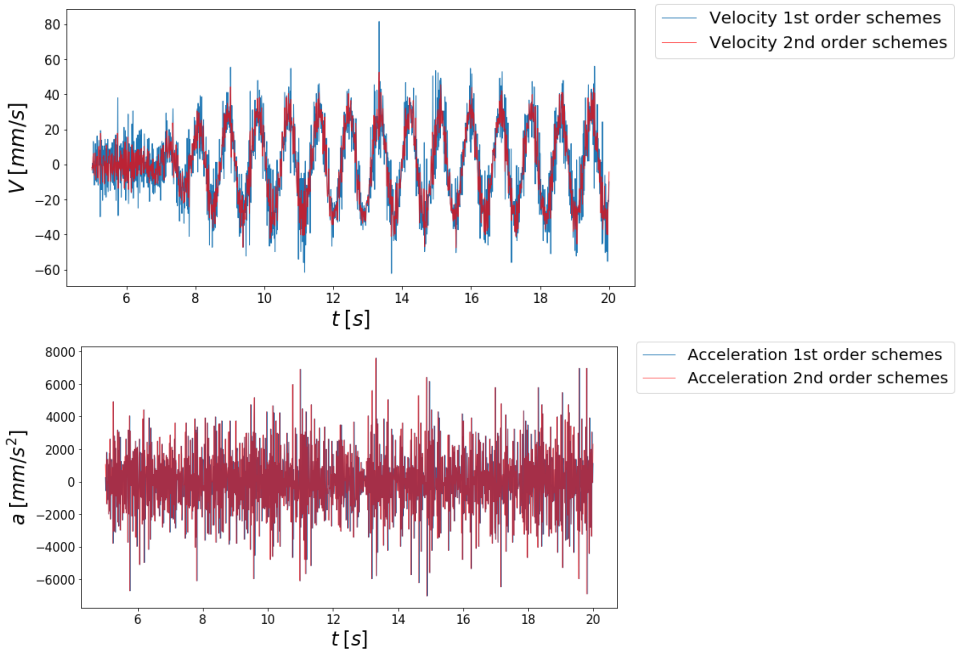
```
fig, ax = plt.subplots(figsize=(10,6))
plt.plot(x, y ,label='Original signal',linewidth=2.0)
plt.xlabel('$t [s]$',fontsize=24)
plt.ylabel('$x [mm]$',fontsize=24)
plt.legend(bbox_to_anchor=(1.02, 1.05), fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.show()
```



First and second order derivative (velocity and acceleration) with first and second order schemes

```
y_prime_1st, y_prime2_1st = Diff_1st_order(x,y) # 1st order schemes
y_prime_2nd, y_prime2_2nd = Diff_2nd_order(x,y) # 2nd order schemes
```

```
fig, ax = plt.subplots(figsize=(12,6))
plt.plot(x,y_prime_1st,label='Velocity 1st order schemes',linewidth=1.0)
plt.plot(x,y_prime_2nd,'r',label='Velocity 2nd order schemes',linewidth=1.0,
alpha=0.7)
plt.xlabel('$t \; [s]$',fontsize=24)
plt.ylabel('$V \; [mm/s]$',fontsize=24)
plt.legend(bbox_to_anchor=(1.02, 1.05), fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.show()

fig, ax = plt.subplots(figsize=(12,6))
plt.plot(x,y_prime2_1st,label='Acceleration 1st order schemes',linewidth=1.0)
plt.plot(x,y_prime2_2nd,'r',label='Acceleration 2nd order schemes',linewidth=1.0,
alpha=0.6)
plt.xlabel('$t \; [s]$',fontsize=24)
plt.ylabel('$a \; [mm/s^2]$',fontsize=24)
plt.legend(bbox_to_anchor=(1.02, 1.05), fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.show()
```



> **Note**
>
> It can already be observed that the derivative operation makes the signal noisier.

Frequency analysis

PSD & FFT Displacement

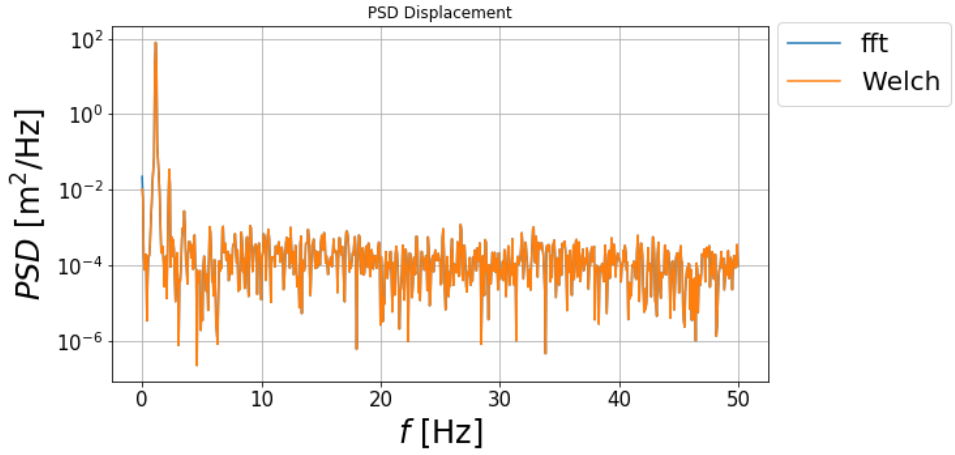
In this section, the frequency content of the signal is analysed by **PSD** (using windowing) and **FFT**.

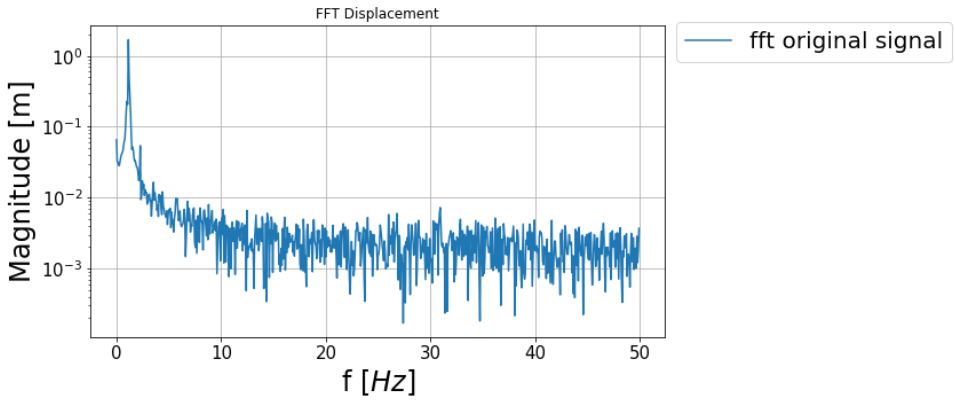
```
dt = 0.01 #s
fs = 1/dt # Hz

Nt = data_raw['Time'].shape[0]
Ntseg = Nt

window = 'hann'
Nwelch = Ntseg

f_welch_x, psd_x = signal.welch(y, fs, window, Nwelch, Nwelch/2, axis=0)
f_fft_x = np.arange(Nt//2+1)/Nt*fs
fft_x = np.fft.rfft(y*get_window(window, Nt))/np.sqrt(fs*(Nt//2+1))*np.sqrt(8./3.)

y_fft=(np.fft.fft(y))
freqs_fft=freq = np.fft.fftfreq(Nt, d=dt)
```

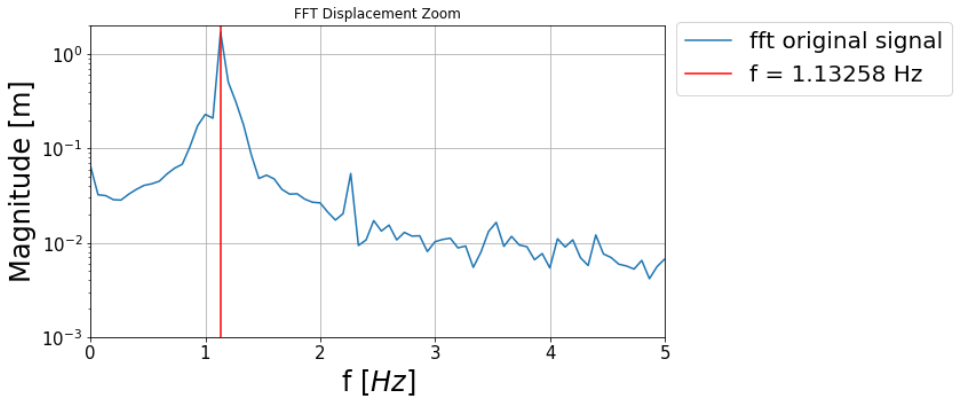
```
A = np.abs(y_fft[:len(y_fft)//2])/Nt
print("The signal has a maximum frequency content at: {}
Hz".format(freqs_fft[np.argmax(A)]))
```

```
The signal has a maximum frequency content at: 1.1325782811459029 Hz
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.semilogy(f_fft_x, np.abs(fft_x)**2,label='fft')
plt.semilogy(f_welch_x, psd_x,label='Welch')
plt.legend(fontsize=20)
plt.grid()
plt.tick_params(axis='both', which='major', labelsize=15)
plt.title('PSD Displacement')
plt.xlabel(r'$f$ [Hz]',fontsize=24)
plt.ylabel(r'$PSD$ [m$^2$/Hz]',fontsize=24)
plt.legend(bbox_to_anchor=(1.3, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.semilogy(freqs_fft[:len(freqs_fft)//2],np.abs(y_fft[:len(y_fft)//2])/Nt,label=
'fft original signal')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('Magnitude [m]',fontsize=24)
#plt.xlim([0, 50])
plt.title('FFT Displacement')
plt.grid()
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.52, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.semilogy(freqs_fft[:len(freqs_fft)//2],np.abs(y_fft[:len(y_fft)//2])/Nt,label=
'fft original signal')
plt.axvline(x=1.13258, color='red', label='f = 1.13258 Hz')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('Magnitude [m]',fontsize=24)
plt.xlim([0, 5])
plt.ylim([1e-3, 2])
plt.title('FFT Displacement Zoom')
plt.grid()
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.52, 1.05), fontsize=20)
plt.show()
```





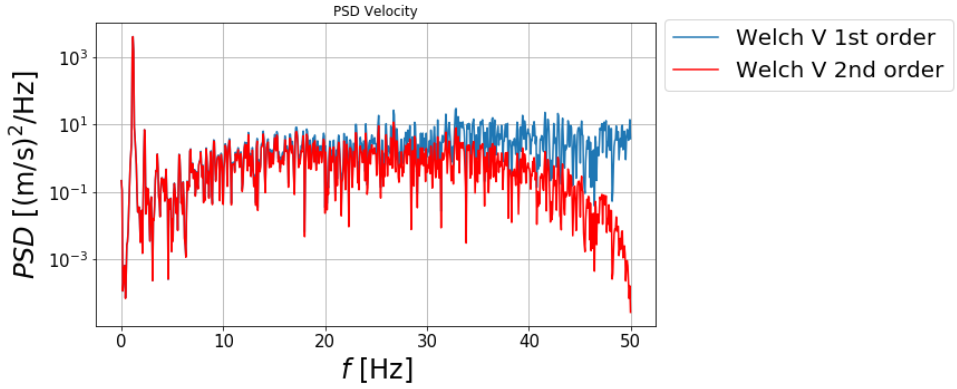


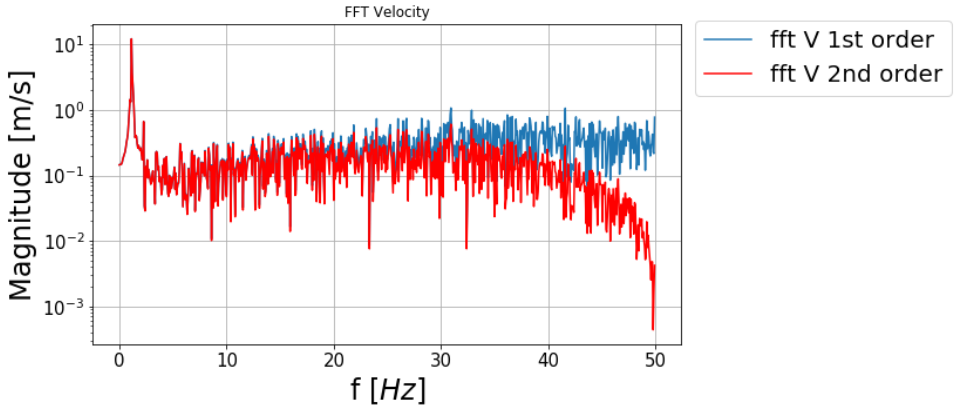
Frequency analysis of the displacement signal shows that the main frequency of motion is 1.13258 Hz. The higher frequency signal components are mainly characterised by noise

PSD & FFT Velocity

```
f_welch_V1, psd_V1 = signal.welch(y_prime_1st, fs, window, Nwelch, Nwelch/2,
axis=0)
f_welch_V2, psd_V2 = signal.welch(y_prime_2nd, fs, window, Nwelch, Nwelch/2,
axis=0)

V1_fft=(np.fft.fft(y_prime_1st))
V2_fft=(np.fft.fft(y_prime_2nd))
V_freqs_fft=freq = np.fft.fftfreq(Nt, d=dt)
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.semilogy(f_welch_V1, psd_V1,label='Welch V 1st order')
plt.semilogy(f_welch_V2, psd_V2,'r',label='Welch V 2nd order')
plt.legend(fontsize=20)
plt.grid()
plt.tick_params(axis='both', which='major', labelsize=15)
plt.title('PSD Velocity')
plt.xlabel(r'$f$ [Hz]',fontsize=24)
plt.ylabel(r'$PSD$ [(m/s)$^2$/Hz]',fontsize=24)
plt.legend(bbox_to_anchor=(1.55, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.semilogy(V_freqs_fft[:len(V_freqs_fft)//2],np.abs(V1_fft[:len(V1_fft)//2])/Nt,
label='fft V 1st order')
plt.semilogy(V_freqs_fft[:len(V_freqs_fft)//2],np.abs(V2_fft[:len(V2_fft)//2])/Nt,
'r',label='fft V 2nd order')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('Magnitude [m/s]',fontsize=24)
plt.title('FFT Velocity')
plt.grid()
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.48, 1.05), fontsize=20)
plt.show()
```

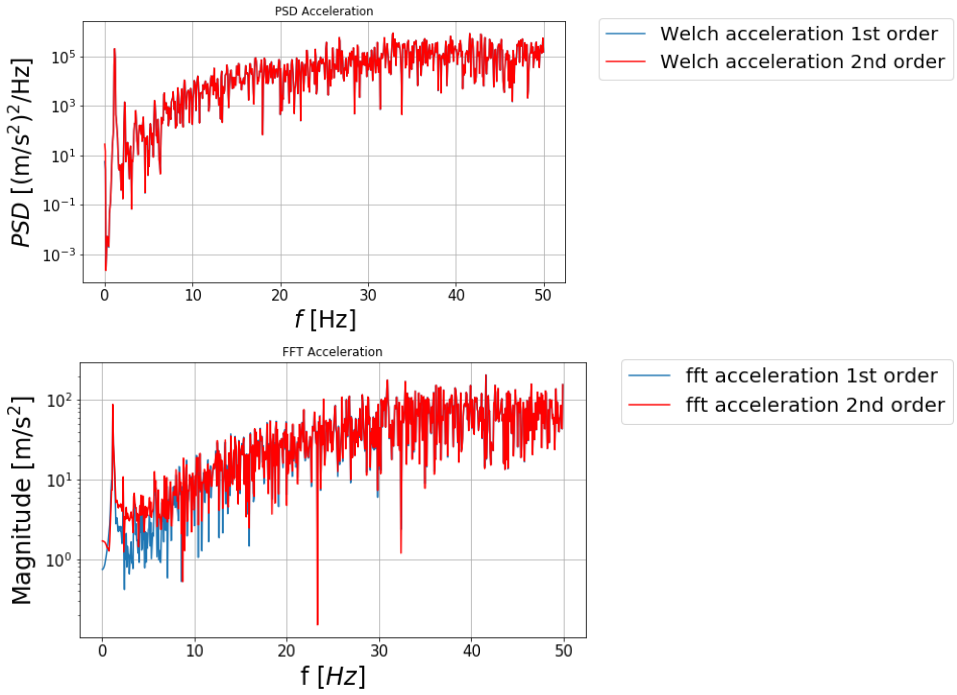




PSD & FFT Acceleration

```
# PSD Acceleration
f_welch_acc1, psd_acc1 = signal.welch(y_prime2_1st, fs, window, Nwelch, Nwelch/2,
axis=0)
f_welch_acc2, psd_acc2 = signal.welch(y_prime2_2nd, fs, window, Nwelch, Nwelch/2,
axis=0)

acc1_fft=(np.fft.fft(y_prime2_1st))
acc2_fft=(np.fft.fft(y_prime2_2nd))
acc_freqs_fft=freq = np.fft.fftfreq(Nt, d=dt)
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.semilogy(f_welch_acc1, psd_acc1,label='Welch acceleration 1st order')
plt.semilogy(f_welch_acc2, psd_acc2,'r',label='Welch acceleration 2nd order')
plt.legend(fontsize=20)
plt.grid()
plt.tick_params(axis='both', which='major', labelsize=15)
plt.title('PSD Acceleration')
plt.xlabel(r'$f$ [Hz]',fontsize=24)
plt.ylabel(r'$PSD$ [(m/s$^2$)$^2$/Hz]',fontsize=24)
plt.legend(bbox_to_anchor=(1.05, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.semilogy(acc_freqs_fft[:len(acc_freqs_fft)//2],np.abs(acc1_fft[:len(acc1_fft)/
/2])/Nt,label='fft acceleration 1st order')
plt.semilogy(acc_freqs_fft[:len(acc_freqs_fft)//2],np.abs(acc2_fft[:len(acc2_fft)/
/2])/Nt,'r',label='fft acceleration 2nd order')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('Magnitude [m/s$^2$]',fontsize=24)
#plt.xlim([0, 50])
plt.title('FFT Acceleration')
plt.grid()
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.05, 1.05), fontsize=20)
plt.show()
```



As noted above, the derivative operation increases the signal noise.
In the case of velocity, the derivative obtained with second-order finite differences has lower frequency content at high frequencies.
The phenomenon of increased signal noise due to the derivative is even more visible in the case of acceleration, where the PSD highlights how the higher frequency has now approximately tha same intensity as the signal containing ones.

Exercise 2: FIR or IIR filters

Use the FIR or IIR filters to control the noise amplification in the differentiation.

Select filter type, order and application (recursive vs convolution) as you think is most appropriate. You can use a non-causal approach to avoid phase lag, and you are free to manipulate the boundaries in any way you consider appropriate.
Comment briefly on what are the main challenges.

```
import numpy as np
from scipy import signal
import pandas as pd
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt, get_window, firwin
from scipy.signal.windows import general_hamming
%matplotlib inline
```

```
def Diff_1st_order(x,y):
    """
    Compute derivatives with first order backward difference approximation:
    y'(xi) = (y(xi)−y(xi−1)) / Δx
    y''(xi) = (y(xi)−2y(xi−1)+y(xi-2) / Δx^2
    For the first point (firsts 2 for the 2nd order derivative),
    The forward difference approximation, that mirrors the expression above,
    is considered
    """

    N = x.shape[0]

    delta_x = np.empty(N)
    y_prime = np.empty(N) # 1st order derivative
    y_prime2 = np.empty(N) # 2nd order derivative
    # Forward first order differencing approx
    delta_x[0] = x[1] - x[0]
    y_prime[0] = (y[1] - y[0]) / delta_x[0]
    y_prime2[0] = (y[2] - 2*y[1] + y[0]) / delta_x[0]**2
    # Backward first order differencing approx
    for i in range(1, N):
        delta_x[i] = x[i] - x[i-1]
        y_prime[i] = (y[i] - y[i-1]) / delta_x[i]
        if i == 1:
            y_prime2[i] = (y[i+2] - 2*y[i+1] + y[i]) / delta_x[i]**2
        else:
            y_prime2[i] = (y[i] - 2*y[i-1] + y[i-2]) / delta_x[i]**2

    return y_prime, y_prime2

def Diff_2nd_order(x,y):
    """
    Compute derivatives with second order central difference approximation:
    y'(xi) = (y(xi+1)−y(xi−1)) / 2Δx
    y''(xi) = (y(xi+1)−2y(xi)+y(xi-1) / Δx^2
    The derivatives for the 2 points at the edge, is computed with the forward
    differencing at the left and the backwards one on the right.
    Forward:
    y'(xi) = (-y(xi+2)+4y(xi+1)-3y(xi)) / 2Δx
    y''(xi) = (-y(xi+3)+4y(xi+2)-5y(xi+1)+2y(xi)) / Δx^2
    Backward:
    y'(xi) = (y(xi-2)-4y(xi-1)+3y(xi)) / 2Δx
    y''(xi) = (-y(xi-3)+4y(xi-2)-5y(xi-1)+2y(xi)) / Δx^2
    """

    N = x.shape[0]

    delta_x = np.empty(N)
    # y_prime = np.gradient(y, x, edge_order=2) # 1st order derivative
    y_prime = np.empty(N) # 1st order derivative
    y_prime2 = np.empty(N) # 2nd order derivative

    # Forward first order differencing approx
    delta_x[0] = x[1] - x[0]
    y_prime[0] = (-y[2] + 4*y[1] - 3*y[0]) / (2*delta_x[0])
    y_prime2[0] = (-y[3] + 4*y[2] - 5*y[1] + 2*y[0]) / delta_x[0]**2
    # Central second order differencing approx
    for i in range(1, N):
        delta_x[i] = x[i] - x[i-1]
        if i == N-1:
            y_prime[i] = (3*y[i] - 4*y[i-1] + y[i-2]) / (2*delta_x[i])
            y_prime2[i] = (-y[i-3] + 4*y[i-2] - 5*y[i-1] + 2*y[i]) / delta_x[i]**2
        else:
            y_prime[i] = (y[i+1] - y[i-1]) / (2*delta_x[i])
            y_prime2[i] = (y[i+1] - 2*y[i] + y[i-1]) / delta_x[i]**2

    return y_prime, y_prime2
```

```
columns = ['Time','ODS_raw']
n = 500
data_raw = pd.read_excel('Signal_raw092.xlsx',
                         header=None, skiprows=n+1, names=columns)
x, y = np.array(data_raw.Time), np.array(data_raw['ODS_raw'])
y_prime_1st, y_prime2_1st = Diff_1st_order(x,y) # 1st order schemes
y_prime_2nd, y_prime2_2nd = Diff_2nd_order(x,y) # 2nd order schemes

dt = 0.01 #s
fs = 1/dt # Hz
Nt = x.shape[0]
```

- **IIR filtering**

  $y_n = \frac{1}{a_0}\left[\sum_{j=1}^{N-1} b_j x_{n-j} - \sum_{k=0}^{N-1} a_k y_{n-k}\right]$

  IIR filters usually require fewer coefficients, are faster and require less memory space with respect to FIR filters.

  Their main disadvantage is their nonlinear phase response and the possibility to become unstable, due to the presence of non zero poles in their transfer function.

- **FIR filtering**

  $y_i = \sum_{k=0}^{N-1} b_k x_{i-k}$

  FIR filters can achieve linear phase response and pass a signal without phase distortion and they are intrinsically stable since they have no poles.

  As drawback, they need a larger number of coefficients respect to IIR filters. For this reason they require more memory and are slower.

> **Tip**
>
> As seen during the SP classes, filtering a signal with FIR filters lead to phase lag. To address this problem we can apply the filter twice (once on the filtered and flipped signal).
> This would lead to BC problems both at the start and the end of the signal. This problem is adressed in the pre-implemented function “filtfilt”

FIR filtering with windowing approach

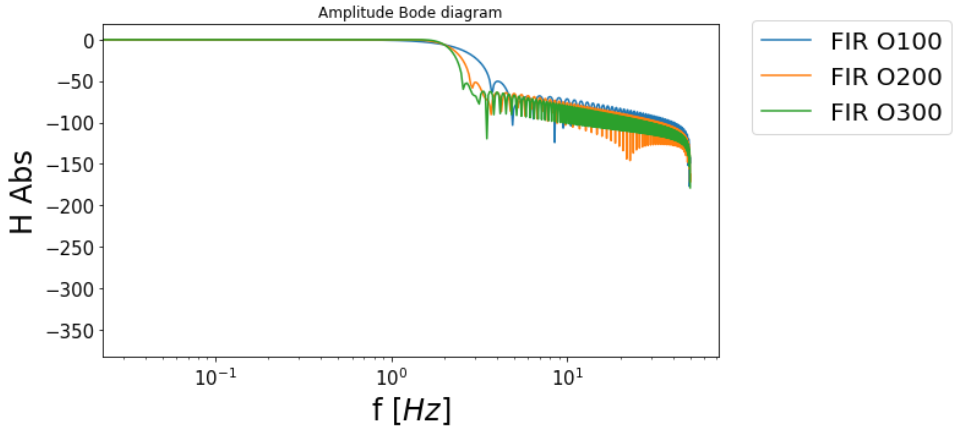
- **Cut-off frequency** has been chosen to be 2 Hz
- FIR filters of order 100, 200, 300 are tested

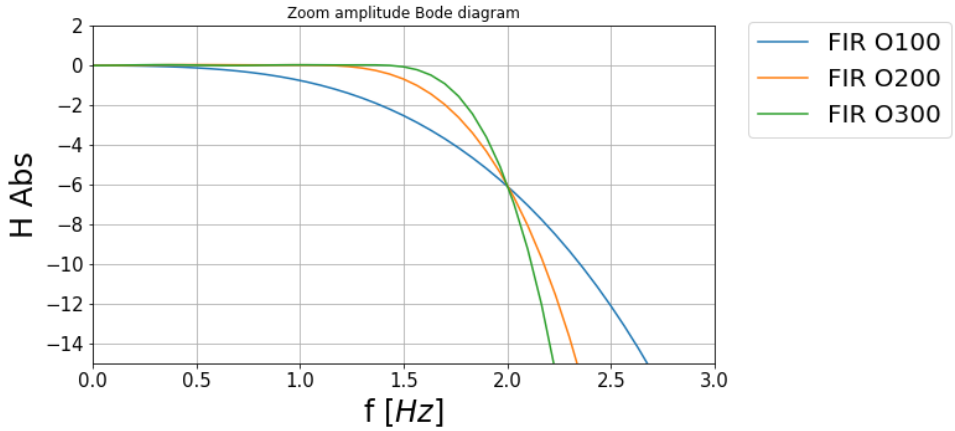
```
f_c = 2 #Hz Cut-off frequency

N_O=100 # this is the order of the filter
h_impulse_100 = firwin(N_O, f_c/fs*2, window='hamming')
N_O=200
h_impulse_200 = firwin(N_O, f_c/fs*2, window='hamming')
N_O=300
h_impulse_300 = firwin(N_O, f_c/fs*2, window='hamming')
```

> **Note**
>
> Different windows lead to slightly different results. Hann lead to little overshoot

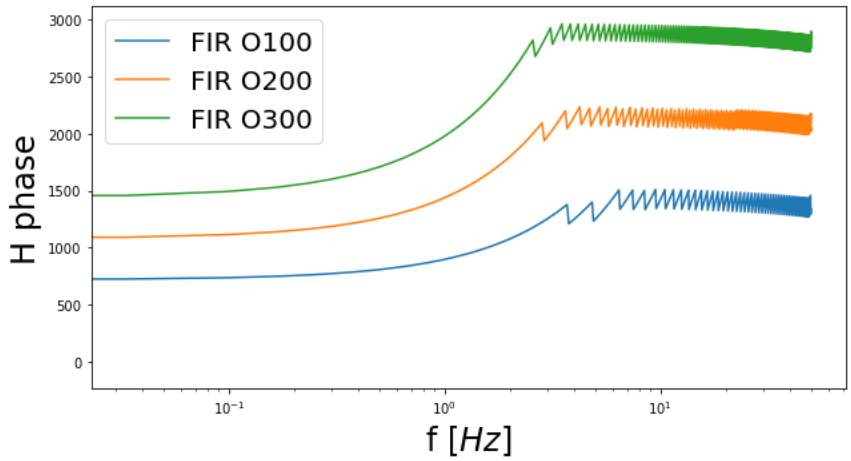
```
n_f = Nt
dtheta= 2*np.pi/n_f

# angular discretization in -pi, pi:
theta_n_2=np.linspace(0,Nt-1, Nt)*dtheta-np.pi

sys_100 = signal.TransferFunction(h_impulse_100, 1, dt=1)
w_FIR_100, mag_FIR_100, phase_FIR_100 = sys_100.bode(w=theta_n_2)

sys_200 = signal.TransferFunction(h_impulse_200, 1, dt=1)
w_FIR_200, mag_FIR_200, phase_FIR_200 = sys_200.bode(w=theta_n_2)

sys_300 = signal.TransferFunction(h_impulse_300, 1, dt=1)
w_FIR_300, mag_FIR_300, phase_FIR_300 = sys_300.bode(w=theta_n_2)
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.semilogx(w_FIR_100/np.pi*fs/2,mag_FIR_100,label='FIR O100')
plt.semilogx(w_FIR_200/np.pi*fs/2,mag_FIR_200,label='FIR O200')
plt.semilogx(w_FIR_300/np.pi*fs/2,mag_FIR_300,label='FIR O300')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('H Abs',fontsize=24)
plt.title('Amplitude Bode diagram')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.4, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.plot(w_FIR_100/np.pi*fs/2,mag_FIR_100,label='FIR O100')
plt.plot(w_FIR_200/np.pi*fs/2,mag_FIR_200,label='FIR O200')
plt.plot(w_FIR_300/np.pi*fs/2,mag_FIR_300,label='FIR O300')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('H Abs',fontsize=24)
plt.xlim([0, 3])
plt.ylim([-15,2])
plt.grid()
plt.title('Zoom amplitude Bode diagram')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.4, 1.05), fontsize=20)
plt.show()
```



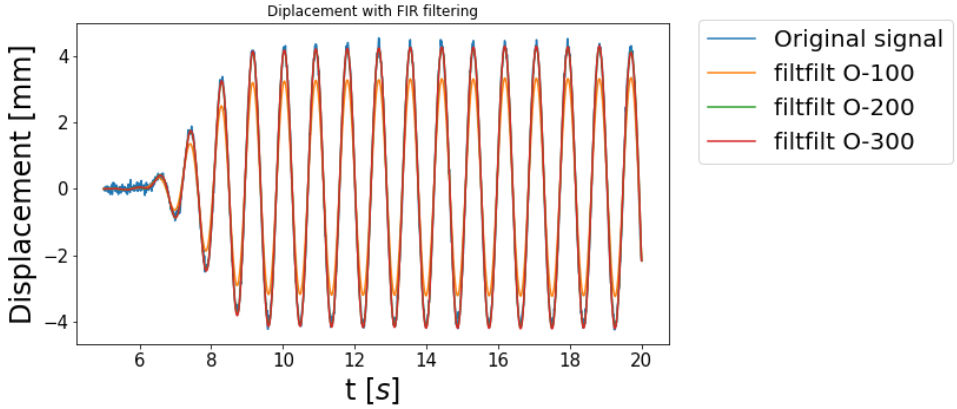


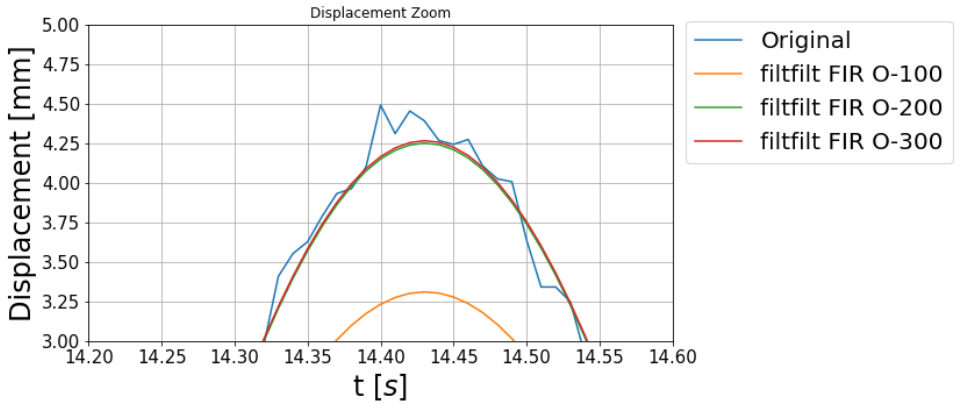

```
# The plot in the classic log vs decibel form are the following:
fig, ax = plt.subplots(figsize=(9,5))
plt.semilogx(w_FIR_100/np.pi*fs/2,phase_FIR_100,label='FIR O100')
plt.semilogx(w_FIR_200/np.pi*fs/2,phase_FIR_200,label='FIR O200')
plt.semilogx(w_FIR_300/np.pi*fs/2,phase_FIR_300,label='FIR O300')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('H phase',fontsize=24)
plt.tight_layout()
plt.show()
```




```
y_filt_filt_100=signal.filtfilt(h_impulse_100,1,y)
y_filt_filt_200=signal.filtfilt(h_impulse_200,1,y)
y_filt_filt_300=signal.filtfilt(h_impulse_300,1,y)
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x,y,label='Original signal')
plt.plot(x,y_filt_filt_100,label='filtfilt O-100')
plt.plot(x,y_filt_filt_200,label='filtfilt O-200')
plt.plot(x,y_filt_filt_300,label='filtfilt O-300')
plt.legend(fontsize=20)
plt.xlabel('t $[s]$',fontsize=24)
plt.ylabel('Displacement [mm]',fontsize=24)
plt.title('Diplacement with FIR filtering')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.5, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x,y,label='Original')
plt.plot(x,y_filt_filt_100,label='filtfilt FIR O-100')
plt.plot(x,y_filt_filt_200,label='filtfilt FIR O-200')
plt.plot(x,y_filt_filt_300,label='filtfilt FIR O-300')
plt.legend(fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlabel('t $[s]$',fontsize=24)
plt.ylabel('Displacement [mm]',fontsize=24)
plt.xlim([14.2, 14.6])
plt.ylim([3, 5])
plt.grid()
plt.title('Displacement Zoom')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.5, 1.05), fontsize=20)
plt.show()
```





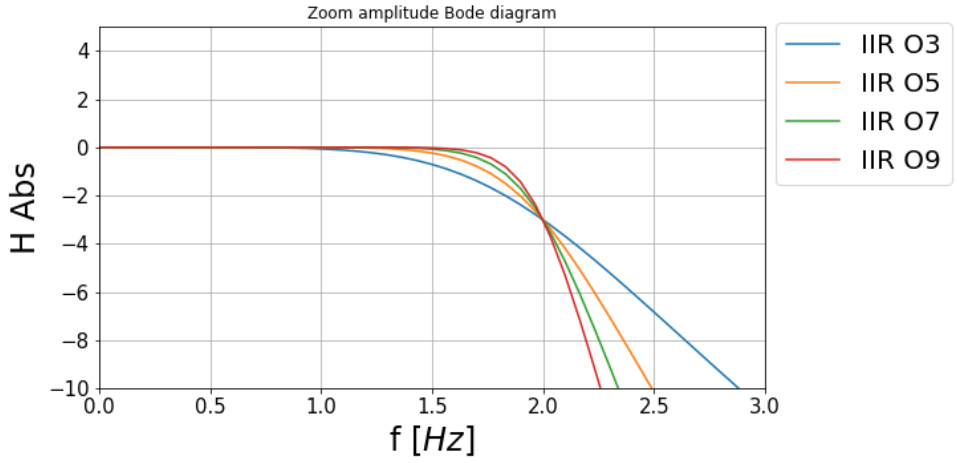
The use of the 'filtfilt' function solves the problem of boundary conditions and phase delay.
The FIR filter of order 100 results in a decrease in signal amplitude.
The filters of order 200 and 300 have very similar performance. The filter of order 200 is chosen because it is faster and less expensive in terms of memory.

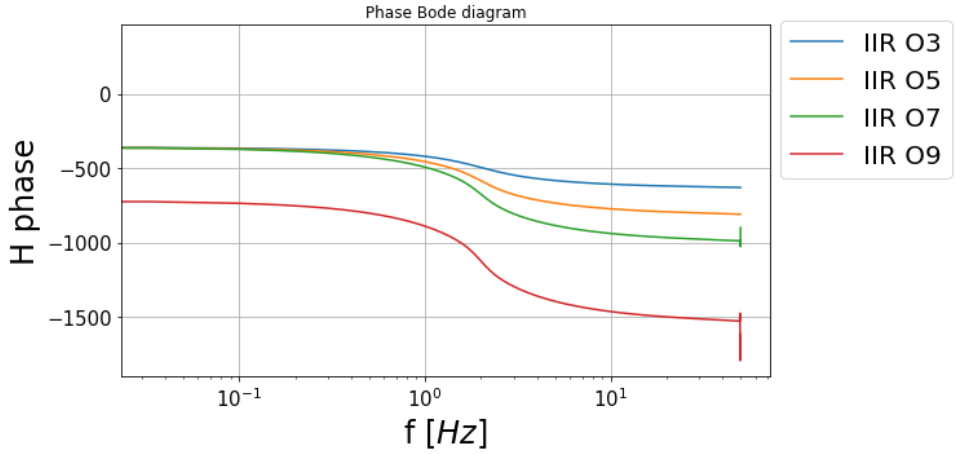
IIR filtering with Butterworth filter

- **Cut-off frequency** has been chosen to be 2 Hz
- IIR filters of order 3, 5, 7, 9 has been tested

```
f_c_IIR = 2 #Hz
b3, a3 = signal.butter(3, f_c_IIR, 'lp', analog=False,fs=fs)
sys = signal.TransferFunction(b3, a3, dt=1)
# Note that the result is in db by default !
w3, mag3, phase3 = sys.bode(w=theta_n_2)

b5, a5 = signal.butter(5, f_c_IIR, 'lp', analog=False,fs=fs)
sys = signal.TransferFunction(b5, a5, dt=1)
# Note that the result is in db by default !
w5, mag5, phase5 = sys.bode(w=theta_n_2)

b7, a7 = signal.butter(7, f_c_IIR, 'lp', analog=False,fs=fs)
sys = signal.TransferFunction(b7, a7, dt=1)
# Note that the result is in db by default !
w7, mag7, phase7 = sys.bode(w=theta_n_2)

b9, a9 = signal.butter(9, f_c_IIR, 'lp', analog=False,fs=fs)
sys = signal.TransferFunction(b9, a9, dt=1)
# Note that the result is in db by default !
w9, mag9, phase9 = sys.bode(w=theta_n_2)
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.plot(w3/np.pi*fs/2,mag3,label='IIR O3')
plt.plot(w3/np.pi*fs/2,mag5,label='IIR O5')
plt.plot(w3/np.pi*fs/2,mag7,label='IIR O7')
plt.plot(w3/np.pi*fs/2,mag9,label='IIR O9')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('H Abs',fontsize=24)
plt.title('Zoom amplitude Bode diagram')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.3, 1.05), fontsize=20)
plt.xlim([0,3])
plt.ylim([-10,5])
plt.grid()
plt.show()


fig, ax = plt.subplots(figsize=(9,5))
plt.semilogx(w3/np.pi*fs/2,phase3,label='IIR O3')
plt.semilogx(w3/np.pi*fs/2,phase5,label='IIR O5')
plt.semilogx(w3/np.pi*fs/2,phase7,label='IIR O7')
plt.semilogx(w3/np.pi*fs/2,phase9,label='IIR O9')
plt.legend(fontsize=20)
plt.xlabel('f $[Hz]$',fontsize=24)
plt.ylabel('H phase',fontsize=24)
plt.title('Phase Bode diagram')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.3, 1.05), fontsize=20)
plt.grid()
plt.show()
```



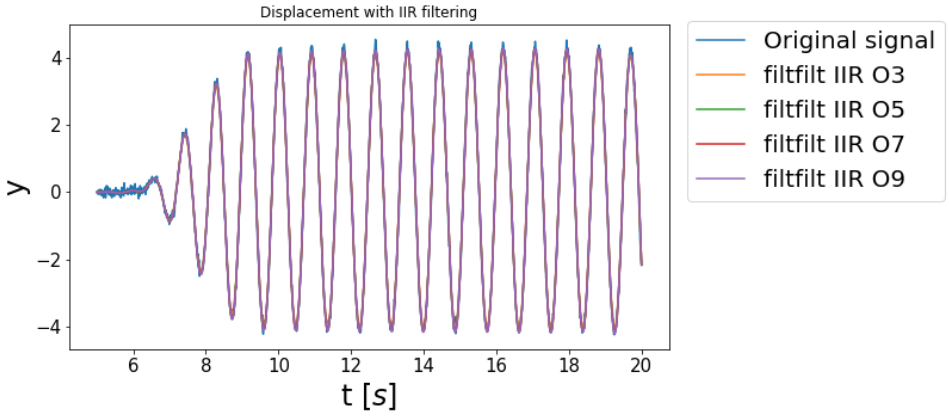


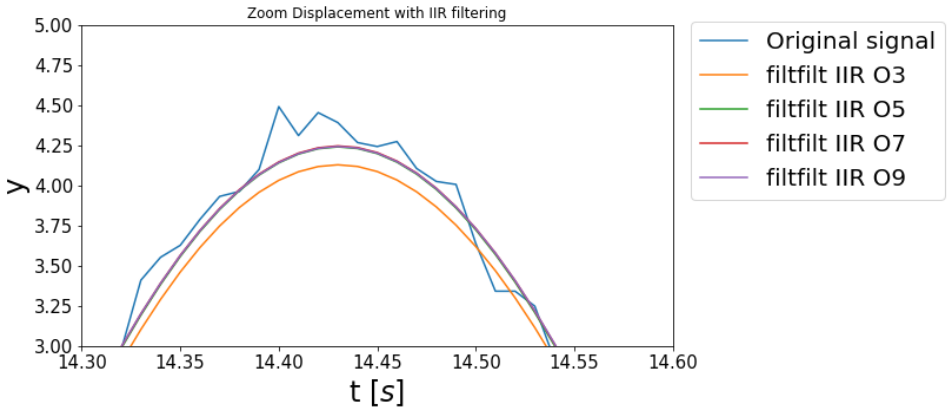
Tip

The $padlen$ argument of the $filtfilt$ function is set to 300 to solve boundary condition problems

```
y_filt_filt_O3=signal.filtfilt(b3,a3,y, padlen=300)
y_filt_filt_O5=signal.filtfilt(b5,a5,y, padlen=300)
y_filt_filt_O7=signal.filtfilt(b7,a7,y, padlen=300)
y_filt_filt_O9=signal.filtfilt(b9,a9,y, padlen=300)
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x,y,label='Original signal')
plt.plot(x,y_filt_filt_O3,label='filtfilt IIR O3')
plt.plot(x,y_filt_filt_O5,label='filtfilt IIR O5')
plt.plot(x,y_filt_filt_O7,label='filtfilt IIR O7')
plt.plot(x,y_filt_filt_O9,label='filtfilt IIR O9')
plt.legend(fontsize=20)
plt.xlabel('t $[s]$',fontsize=24)
plt.ylabel('y',fontsize=24)
plt.title('Displacement with IIR filtering')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.01, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x,y,label='Original signal')
plt.plot(x,y_filt_filt_O3,label='filtfilt IIR O3')
plt.plot(x,y_filt_filt_O5,label='filtfilt IIR O5')
plt.plot(x,y_filt_filt_O7,label='filtfilt IIR O7')
plt.plot(x,y_filt_filt_O9,label='filtfilt IIR O9')
plt.legend(fontsize=20)
plt.xlabel('t $[s]$',fontsize=24)
plt.ylabel('y',fontsize=24)
plt.xlim([14.3, 14.6])
plt.ylim([3, 5])
plt.title('Zoom Displacement with IIR filtering')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.48, 1.05), fontsize=20)
plt.show()
```



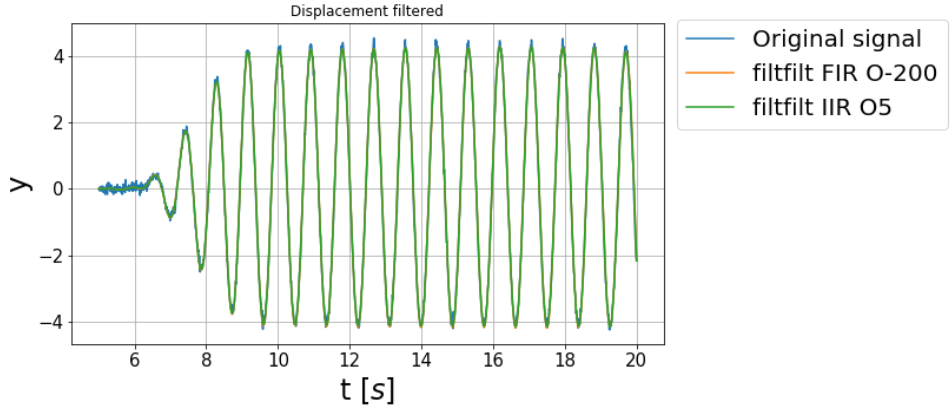


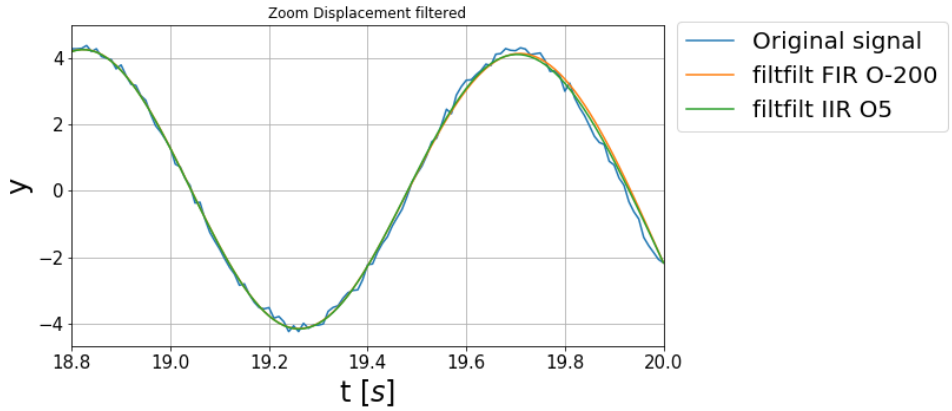
The original signal filtered with IIR filters of order 5,7,9 leads to almost indistinguishable lines.

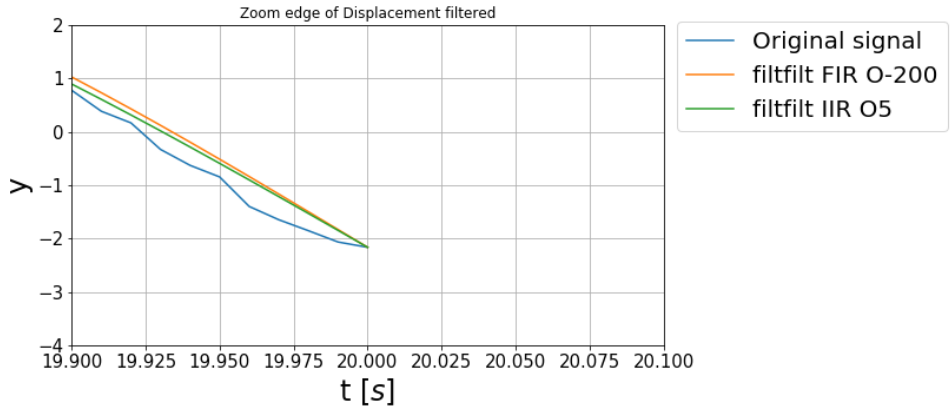
The filter of order 5 is chosen because the bode diagrams show no signs of oscillations

Comparison IIR and FIR filtering

```
fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x,y,label='Original signal')
plt.plot(x,y_filt_filt_200,label='filtfilt FIR O-200')
plt.plot(x,y_filt_filt_O5,label='filtfilt IIR O5')
plt.legend(fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlabel('t $[s]$',fontsize=24)
plt.ylabel('y',fontsize=24)
plt.grid()
plt.title('Displacement filtered')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.5, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x,y,label='Original signal')
plt.plot(x,y_filt_filt_200,label='filtfilt FIR O-200')
plt.plot(x,y_filt_filt_O5,label='filtfilt IIR O5')
plt.legend(fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlabel('t $[s]$',fontsize=24)
plt.ylabel('y',fontsize=24)
# plt.xlim([14.3, 14.6])
# plt.ylim([3, 5])
plt.xlim([18.8, 20])
# plt.ylim([3, 5])
plt.grid()
plt.title('Zoom Displacement filtered')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.5, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x,y,label='Original signal')
plt.plot(x,y_filt_filt_200,label='filtfilt FIR O-200')
plt.plot(x,y_filt_filt_O5,label='filtfilt IIR O5')
plt.legend(fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlabel('t $[s]$',fontsize=24)
plt.ylabel('y',fontsize=24)
plt.xlim([19.9, 20.1])
plt.ylim([-4, 2])
plt.grid()
plt.title('Zoom edge of Displacement filtered')
plt.tick_params(axis='both', which='major', labelsize=15)
plt.legend(bbox_to_anchor=(1.5, 1.05), fontsize=20)
plt.show()
```







The main challenges in constructing a filter are many.

- **The choice of cut-off frequency.**
  In order not to risk decreasing the amplitude of the signal of interest, one might consider moving the cut-off frequency further away from the frequency of interest. This would entail accepting that more noisy frequencies remain unattenuated in the signal.
- **The choice of filter order:**
  In both cases, IIR and FIR, a higher order filter leads to a higher phase delay (in the case of IIR also to the presence of oscillations) but also to a higher filter slope near the cut-off frequency.
- **The problem of dealing with boundary conditions:**
  A causal approach (recursive (FIR or IIR) or zero-padding+convolution (FIR)) leads to phase delay and boundary problems for the first instants of the signal. A non-causal approach (convolution (FIR)) eliminates the phase delay problem but brings boundary problems to the final instants of the signal.
  In this case, the filtfilt function was adopted, which after mirroring the signal applies a recursive formula successively in one direction and in the other to eliminate the phase delay. Then the signal is truncated to the initial length to eliminate the boundary problems.

Exercise 3: Derivatives in Fourier domain

Compute derivatives in the Fourier Domain.

Computing derivatives in the frequency domain is equivalent to performing multiplications. This is the essence of spectral and pseudospectral methods in CFD. Show it using the DFT definition and then using Python. Is this true everywhere?

Compute velocity and acceleration using spectral differentiation + filtering.

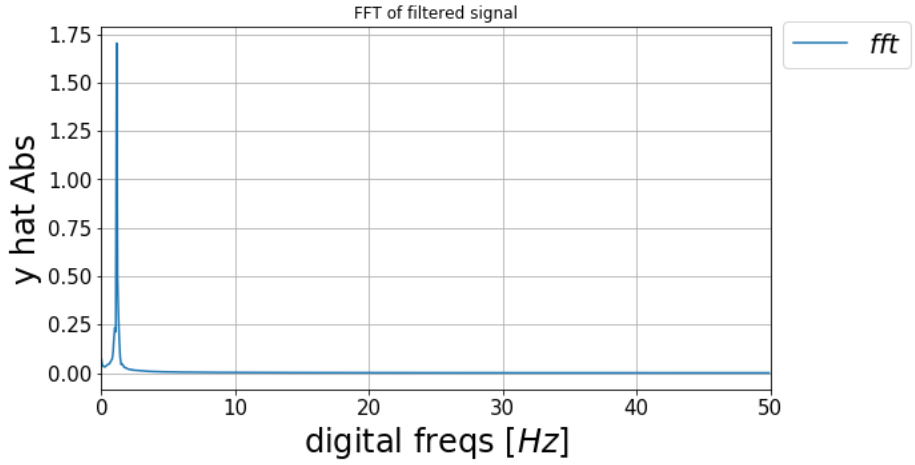
```
import numpy as np
from scipy import signal
import pandas as pd
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt, get_window, firwin
from scipy.signal.windows import general_hamming
%matplotlib inline
```

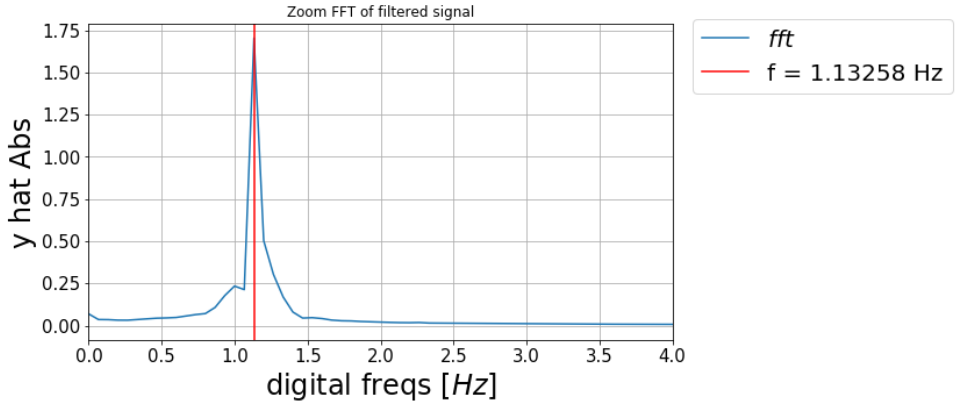
```
def Diff_1st_order(x,y):
    """
    Compute derivatives with first order backward difference approximation:
    y'(xi) = (y(xi)−y(xi−1)) / Δx
    y''(xi) = (y(xi)−2y(xi−1)+y(xi-2) / Δx^2
    For the first point (firsts 2 for the 2nd order derivative),
    The forward difference approximation, that mirrors the expression above,
    is considered
    """

    N = x.shape[0]

    delta_x = np.empty(N)
    y_prime = np.empty(N) # 1st order derivative
    y_prime2 = np.empty(N) # 2nd order derivative
    # Forward first order differencing approx
    delta_x[0] = x[1] - x[0]
    y_prime[0] = (y[1] - y[0]) / delta_x[0]
    y_prime2[0] = (y[2] - 2*y[1] + y[0]) / delta_x[0]**2
    # Backward first order differencing approx
    for i in range(1, N):
        delta_x[i] = x[i] - x[i-1]
        y_prime[i] = (y[i] - y[i-1]) / delta_x[i]
        if i == 1:
            y_prime2[i] = (y[i+2] - 2*y[i+1] + y[i]) / delta_x[i]**2
        else:
            y_prime2[i] = (y[i] - 2*y[i-1] + y[i-2]) / delta_x[i]**2

    return y_prime, y_prime2

def Diff_2nd_order(x,y):
    """
    Compute derivatives with second order central difference approximation:
    y'(xi) = (y(xi+1)−y(xi−1)) / 2Δx
    y''(xi) = (y(xi+1)−2y(xi)+y(xi-1) / Δx^2
    The derivatives for the 2 points at the edge, is computed with the forward
    differencing at the left and the backwards one on the right.
    Forward:
    y'(xi) = (-y(xi+2)+4y(xi+1)-3y(xi)) / 2Δx
    y''(xi) = (-y(xi+3)+4y(xi+2)-5y(xi+1)+2y(xi)) / Δx^2
    Backward:
    y'(xi) = (y(xi-2)-4y(xi-1)+3y(xi)) / 2Δx
    y''(xi) = (-y(xi-3)+4y(xi-2)-5y(xi-1)+2y(xi)) / Δx^2
    """

    N = x.shape[0]

    delta_x = np.empty(N)
    # y_prime = np.gradient(y, x, edge_order=2) # 1st order derivative
    y_prime = np.empty(N) # 1st order derivative
    y_prime2 = np.empty(N) # 2nd order derivative

    # Forward first order differencing approx
    delta_x[0] = x[1] - x[0]
    y_prime[0] = (-y[2] + 4*y[1] - 3*y[0]) / (2*delta_x[0])
    y_prime2[0] = (-y[3] + 4*y[2] - 5*y[1] + 2*y[0]) / delta_x[0]**2
    # Central second order differencing approx
    for i in range(1, N):
        delta_x[i] = x[i] - x[i-1]
        if i == N-1:
            y_prime[i] = (3*y[i] - 4*y[i-1] + y[i-2]) / (2*delta_x[i])
            y_prime2[i] = (-y[i-3] + 4*y[i-2] - 5*y[i-1] + 2*y[i]) / delta_x[i]**2
        else:
            y_prime[i] = (y[i+1] - y[i-1]) / (2*delta_x[i])
            y_prime2[i] = (y[i+1] - 2*y[i] + y[i-1]) / delta_x[i]**2

    return y_prime, y_prime2
```

```
columns = ['Time','ODS_raw']
n = 500
data_raw = pd.read_excel('Signal_raw092.xlsx',
                         header=None, skiprows=n+1, names=columns)
x, y = np.array(data_raw.Time), np.array(data_raw['ODS_raw'])
y_prime_1st, y_prime2_1st = Diff_1st_order(x,y) # 1st order schemes
y_prime_2nd, y_prime2_2nd = Diff_2nd_order(x,y) # 2nd order schemes

dt = 0.01 #s
fs = 1/dt # Hz
f_c = 2 #Hz Cut-off frequency
Nt = x.shape[0]


N_O=200 # this is the order of the filter
h_impulse_200 = firwin(N_O, f_c/fs*2, window='hamming')
y_filt_filt_200=signal.filtfilt(h_impulse_200,1,y)
```

```
y_filt_fft=(np.fft.fft(y_filt_filt_200))
freqs_fft=freq = np.fft.fftfreq(Nt, d=dt)
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.plot(freqs_fft[:len(freqs_fft)//2],np.abs(y_filt_fft[:len(y_filt_fft)//2])/Nt,
label='$fft$')
plt.legend(fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlabel('digital freqs $[Hz]$',fontsize=24)
plt.ylabel('y hat Abs',fontsize=24)
plt.xlim([0, 50])
plt.grid()
plt.title('FFT of filtered signal')
plt.legend(bbox_to_anchor=(1.23, 1.05), fontsize=20)
plt.show()

fig, ax = plt.subplots(figsize=(9,5))
plt.plot(freqs_fft[:len(freqs_fft)//2],np.abs(y_filt_fft[:len(y_filt_fft)//2])/Nt,
label='$fft$')
plt.axvline(x=1.13258, color='red', label='f = 1.13258 Hz')
plt.legend(fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlabel('digital freqs $[Hz]$',fontsize=24)
plt.ylabel('y hat Abs',fontsize=24)
plt.xlim([0, 4])
plt.grid()
plt.title('Zoom FFT of filtered signal')
plt.legend(bbox_to_anchor=(1.5, 1.05), fontsize=20)
plt.show()
```





Demonstration derivatives in Fourier domain

From the definition of **inverse Fourier transform**, we have:

$y(t) = \frac{1}{T}\sum_{k=0}^{N-1} Y_k e^{i\frac{2\pi}{T}kt}$

Where Y_k is the Fourier transform of the signal.

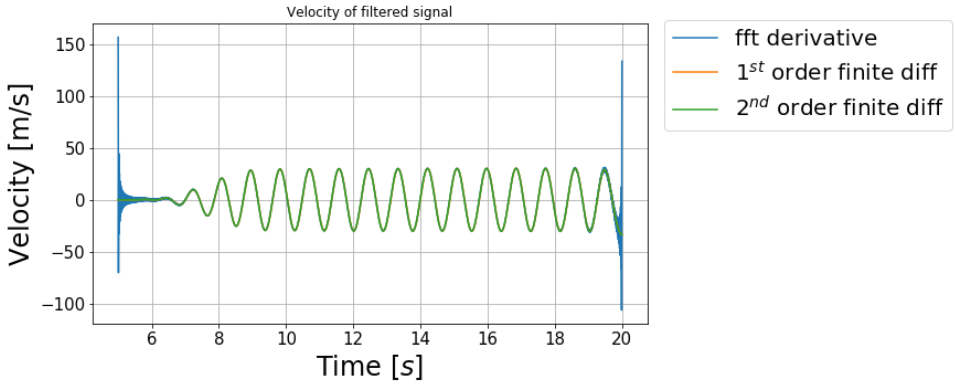
Taking time differentiation on both sides, we get:

$\frac{d}{dt}y(t) = \frac{d}{dt}\left[\frac{1}{T}\sum_{k=0}^{N-1} Y_k e^{i\frac{2\pi}{T}kt}\right]$

$\Rightarrow y'(t) = \frac{1}{T}\sum_{k=0}^{N-1} i\frac{2\pi}{T}kY_k e^{i\frac{2\pi}{T}kt}$

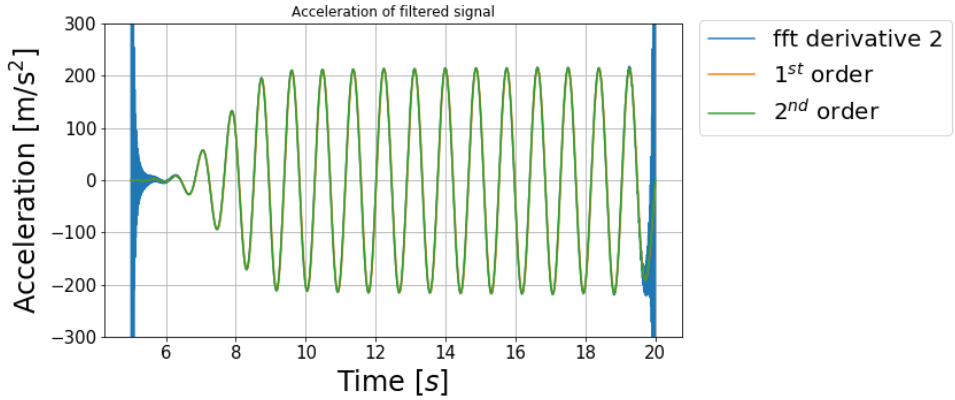
```
kappa = np.linspace(0,Nt-1,Nt) - (Nt-1)/2
kappa = np.fft.fftshift(kappa)
kappa *= 2 * np.pi / (x[-1]-x[0])
y_filt_fft_prime = 1j * kappa * y_filt_fft
dfFFT = np.real(np.fft.ifft(y_filt_fft_prime))

y_filt_prime_1st, y_filt_prime2_1st = Diff_1st_order(x,y_filt_filt_200)
y_filt_prime_2nd, y_filt_prime2_2nd = Diff_2nd_order(x,y_filt_filt_200)
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x, dfFFT.real,label='fft derivative')
plt.plot(x, y_filt_prime_1st,label='$1^{st}$ order finite diff')
plt.plot(x, y_filt_prime_2nd,label='$2^{nd}$ order finite diff')
plt.legend(fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlabel('Time $[s]$',fontsize=24)
plt.ylabel('Velocity [m/s]',fontsize=24)
plt.grid()
plt.title('Velocity of filtered signal')
plt.legend(bbox_to_anchor=(1.01, 1.05), fontsize=20)
plt.show()
```




```
y_filt_fft_prime2 = -kappa**2 * y_filt_fft
ddfFFT = np.real(np.fft.ifft(y_filt_fft_prime2))
```

```
fig, ax = plt.subplots(figsize=(9,5))
plt.plot(x, ddfFFT.real,label='fft derivative 2')
plt.plot(x, y_filt_prime2_1st,label='$1^{st}$ order')
plt.plot(x, y_filt_prime2_2nd,label='$2^{nd}$ order')
plt.legend(fontsize=20)
plt.tick_params(axis='both', which='major', labelsize=15)
plt.xlabel('Time $[s]$',fontsize=24)
plt.ylabel('Acceleration [m/s$^2$]',fontsize=24)
plt.ylim([-300, 300])
# plt.xlim([19.9, 20])
plt.grid()
plt.title('Acceleration of filtered signal')
plt.legend(bbox_to_anchor=(1.01, 1.05), fontsize=20)
plt.show()
```



Derivatives obtained in the frequency domain obtain results in line with finite differences on the filtered signal. Oscillation are present at the extremes due to the fact that the Fourier transform is based on the assumptions of a **periodic and infinite** signal. Assumptions that are not fulfilled in this case

By Umberto Zucchelli