2_TimeAndFrequencyDomain_template

April 19, 2020

1 Load system matrices as sparse-matrices

Here we use the sparse module of scipy. The module contains functions for linear algebra with sparse matrices (scipy.sparse.linalg). Do **not** mix with numpy functions! Convert them to dense arrays using `.toarray()` if you need numpy.

Read system matrices as sparse matrices like this

```
from scipy.io import mmread
from scipy.sparse import csc_matrix
M = csc_matrix(mmread('Ms.mtx')) # mass matrix
K = csc_matrix(mmread('Ks.mtx')) # stiffness matrix
C = csc_matrix(K.shape) # a zeros damping matrix
X = mmread('X.mtx') # coodinate matrix with columns corresponding to x,y,z position of the nod
N = X.shape[0] # number of nodes
```

[28]:
```
from scipy.io import mmread
from scipy.sparse import csc_matrix
from scipy.sparse.linalg import eigsh
from scipy.sparse.linalg import inv

import numpy as np

import matplotlib as matplot
import matplotlib.pyplot as plt
matplot.rcParams.update({'figure.max_open_warning': 0})

# Uncomment the following line and edit the path to ffmpeg if you want to write␣
 ↪the video files!
#plt.rcParams['animation.ffmpeg_path'] ='N:\\Applications\\ffmpeg\\bin\\ffmpeg.
 ↪exe'

from mpl_toolkits.mplot3d import Axes3D

import sys
np.set_printoptions(threshold=sys.maxsize)

from numpy.fft import rfft, rfftfreq
```

```python
from utility_functions import Newmark
```

[3]:
```python
M = csc_matrix(mmread('Ms.mtx')) # mass matrix
K = csc_matrix(mmread('Ks.mtx')) # stiffness matrix
C = csc_matrix(K.shape) # a zeros damping matrix
X = mmread('X.mtx') # coodinate matrix with columns corresponding to x,y,z␣
↪position of the nodes

N = X.shape[0] # number of nodes

nprec = 6 # precision for finding uniqe values

# get grid vectors (the unique vectors of the x,y,z coodinate-grid)
x = np.unique(np.round(X[:,0],decimals=nprec))
y = np.unique(np.round(X[:,1],decimals=nprec))
z = np.unique(np.round(X[:,2],decimals=nprec))

# grid matrices
Xg = np.reshape(X[:,0],[len(y),len(x),len(z)])
Yg = np.reshape(X[:,1],[len(y),len(x),len(z)])
Zg = np.reshape(X[:,2],[len(y),len(x),len(z)])
```

2 Select nodes for application of boundary conditions and loads

We want to find all indices for nodes located on the edge of the plate. To find all nodes on one edge we search for nodes with e.g. coordinates sufficiently close (numerical tolerance) to the minimum y-coordinate (south edge) . Repeating this for all sides gives all edge nodes.

```python
tol = 1e-12

# constrain all edges
Nn = np.argwhere(np.abs(X[:,1]-X[:,1].max())<tol).ravel() # Node indices of N-Edge nodes
No = np.argwhere(np.abs(X[:,0]-X[:,0].max())<tol).ravel() # Node indices of O-Edge nodes
Ns = np.argwhere(np.abs(X[:,1]-X[:,1].min())<tol).ravel() # Node indices of S-Edge nodes
Nw = np.argwhere(np.abs(X[:,0]-X[:,0].min())<tol).ravel() # Node indices of W-Edge nodes

Nnosw = np.hstack([Nn,No,Ns,Nw])
```

[4]:
```python
tol = 1e-12

# constrain all edges
Nn = np.argwhere(np.abs(X[:,1]-X[:,1].max())<tol).ravel() # Node indices of␣
↪N-Edge nodes
No = np.argwhere(np.abs(X[:,0]-X[:,0].max())<tol).ravel() # Node indices of␣
↪O-Edge nodes
Ns = np.argwhere(np.abs(X[:,1]-X[:,1].min())<tol).ravel() # Node indices of␣
↪S-Edge nodes
```

```
Nw = np.argwhere(np.abs(X[:,0]-X[:,0].min())<tol).ravel() # Node indices of␣
↪W-Edge nodes

Nnosw = np.unique(np.concatenate((Nn,No,Ns,Nw))) #concatenate all and only take␣
↪unique (remove the double ones)
```

We can also search for the closest point to a particular location.

```
P1 = [0.2,0.12,0.003925]
N1 = np.argmin(np.sum((X-P1)**2,axis=1))
P2 = [0.0,-0.1,0.003925]
N2 = np.argmin(np.sum((X-P2)**2,axis=1))
```

of for all node on the top of the plate

```
Nt = np.argwhere(np.abs(X[:,2]-X[:,2].max())<tol)[:,0]
```

[5]:
```
P1 = [0.2,0.12,0.003925]
N1 = np.argmin(np.sum((X-P1)**2,axis=1))
P2 = [0.0,-0.1,0.003925]
N2 = np.argmin(np.sum((X-P2)**2,axis=1))

Nt = np.argwhere(np.abs(X[:,2]-X[:,2].max())<tol).ravel()
```

3 Constrain the system

We apply the clamping boundary condition on all edges by eliminating all degrees of freedom of the edge nodes from the system matrices.

```
# indices of x, y, and z DoFs in the global system
# can be used to get DoF-index in global system, e.g. for y of node n by Iy[n]
Ix = np.arange(N)*3 # index of x-dofs
Iy = np.arange(N)*3+1
Iz = np.arange(N)*3+2

# select which indices in the global system must be constrained
If = np.array([Ix[Nnosw],Iy[Nnosw],Iz[Nnosw]]).ravel() # dof indices of fix constraint
Ic = np.array([(i in If) for i in np.arange(3*N)]) # boolean array of constraind dofs

# compute the reduced system
Kc = csc_matrix(K[np.ix_(~Ic,~Ic)])
Mc = csc_matrix(M[np.ix_(~Ic,~Ic)])
Cc = csc_matrix(C[np.ix_(~Ic,~Ic)])
```

[6]:
```
# indices of x, y, and z DoFs in the global system
# can be used to get DoF-index in global system, e.g. for y of node n by Iy[n]
Ix = np.arange(N)*3 # index of x-dofs
Iy = np.arange(N)*3+1
Iz = np.arange(N)*3+2
```

```
# select which indices in the global system must be constrained
If = np.array([Ix[Nnosw],Iy[Nnosw],Iz[Nnosw]]).ravel() # dof indices of fix␣
↪constraint
Ic = np.array([(i in If) for i in np.arange(3*N)]) # boolean array of␣
↪constraind dofs

# compute the reduced system
Kc = csc_matrix(K[np.ix_(~Ic,~Ic)])
Mc = csc_matrix(M[np.ix_(~Ic,~Ic)])
Cc = csc_matrix(C[np.ix_(~Ic,~Ic)])
```

4 Compute Natural Frequencies and Mode Shapes

Use the (ARPACK) routines for sparce matrices.

```
from scipy.sparse.linalg import eigsh
```

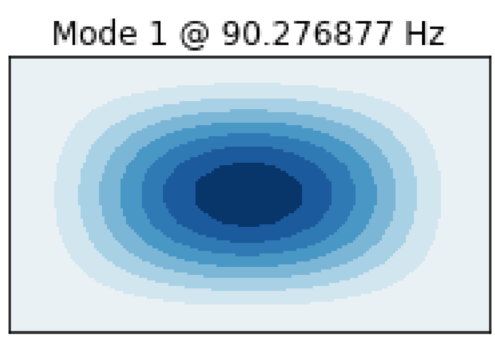
[7]:
```
def plotmodes(V_var,W_var) :
    for i,v in enumerate(V_var.T) : # iterate over eigenvectors
        c = np.reshape(v[Iz[Nt]],[len(y),len(x)])
        lim = np.max(np.abs(c))
        fig,ax = plt.subplots(figsize=[3.5,2])
        ax.contourf(x,y,c,cmap=plt.get_cmap('RdBu'),vmin=-lim,vmax=lim)
        ax.set_aspect('equal')
        ax.set_title('Mode %i @ %f Hz'%(i+1,np.sqrt(abs(W_var[i]))/2/np.pi))
        ax.set_xticks([])
        ax.set_yticks([])
        fig.tight_layout()


def makeFancyModes(V_var,Ic_var,W_var,Ncon) :
    for i,v in enumerate(V_var.T) :
        u = np.zeros(3*N) # initialize displacement vector
        uc = np.real(V_var[:,i]) #without exp. power term, since we only look␣
↪at the static displacment
        u[~Ic_var] = uc

        # plot in 3D
        fig,ax = plt.subplots(subplot_kw={'projection':'3d'})
        ax.scatter(X[:,0],X[:,1],X[:,2],s=5,label='undeformed') # undeformed

        # format U like X
        U = np.array([u[Ix],u[Iy],u[Iz]]).T

        # scale factor for plotting
        s = 0.5/np.max(np.sqrt(np.sum(U**2,axis=0)))
```

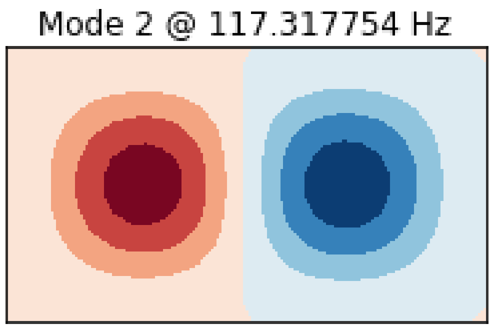
```
        Xu = X + s*U # defomed configuration (displacement scaled by s)

        ax.scatter(Xu[:,0],Xu[:,1],Xu[:,2],s=5,c='g',label='deformed')
        ax.
↪scatter(X[Ncon,0],X[Ncon,1],X[Ncon,2],s=50,marker='x',label='constraint')

        ax.set_title('Mode %i @ %f Hz'%(i+1,np.sqrt(abs(W_var[i]))/2/np.pi))
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')
        ax.legend()
```

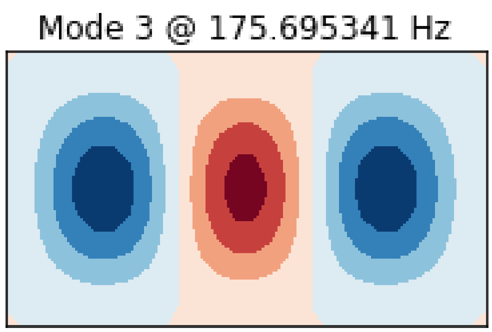
Compute the first 10 modes and plot them.

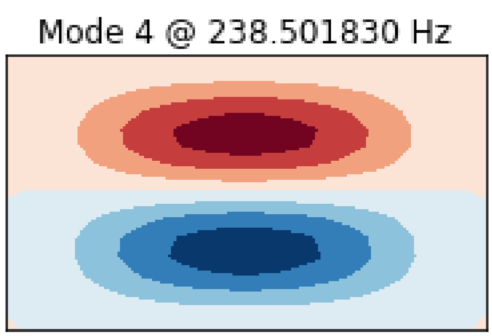
[8]:
```
# only compute a subset of modes of the reduced model
k = 10
W,V = eigsh(Kc,k,Mc,sigma=0,which='LM',maxiter = 1000)
```

[9]:
```
# add missing nodes (constraints)
V_new = np.zeros((len(Iz)*3,k))
If_sort_all = np.sort(If);

for i,v in enumerate(V.T):
    V_dat = V[:,i]
    for d,idx in enumerate(If_sort_all) :
        V_dat = np.insert(V_dat,idx,0)
    V_new[:,i] = V_dat

# do it like the prof suggested
plotmodes(V_new,W)

# # do it fancier
makeFancyModes(V,Ic,W,Nnosw)
```

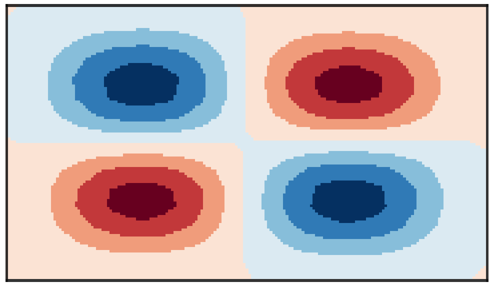



Mode 2 @ 117.317754 Hz

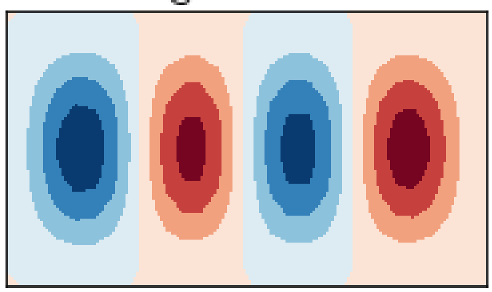

Mode 3 @ 175.695341 Hz


Mode 4 @ 238.501830 Hz

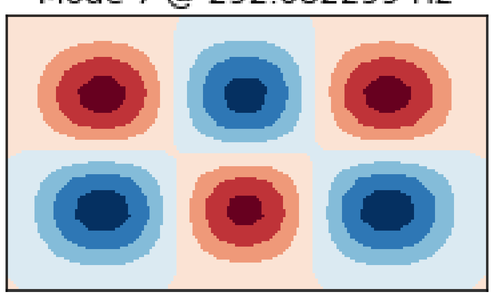
Mode 5 @ 254.410780 Hz



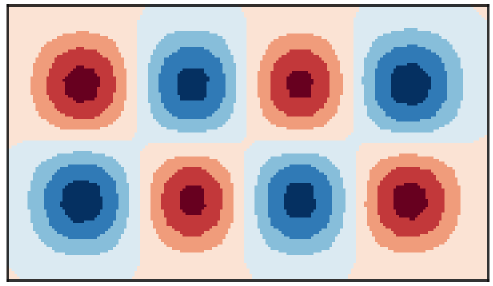
Mode 6 @ 265.083064 Hz



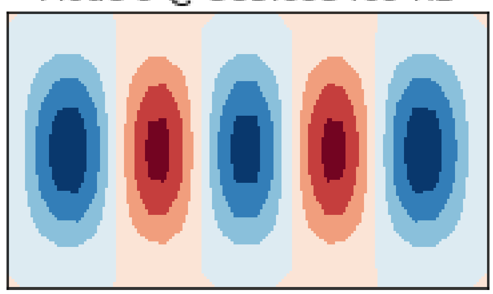
Mode 7 @ 292.082299 Hz



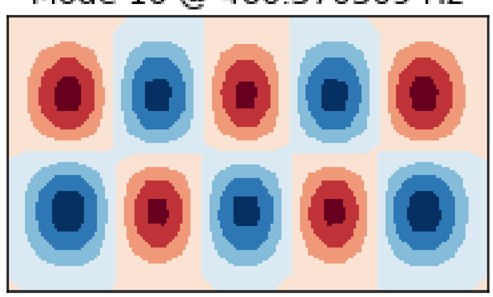
Mode 8 @ 359.702453 Hz

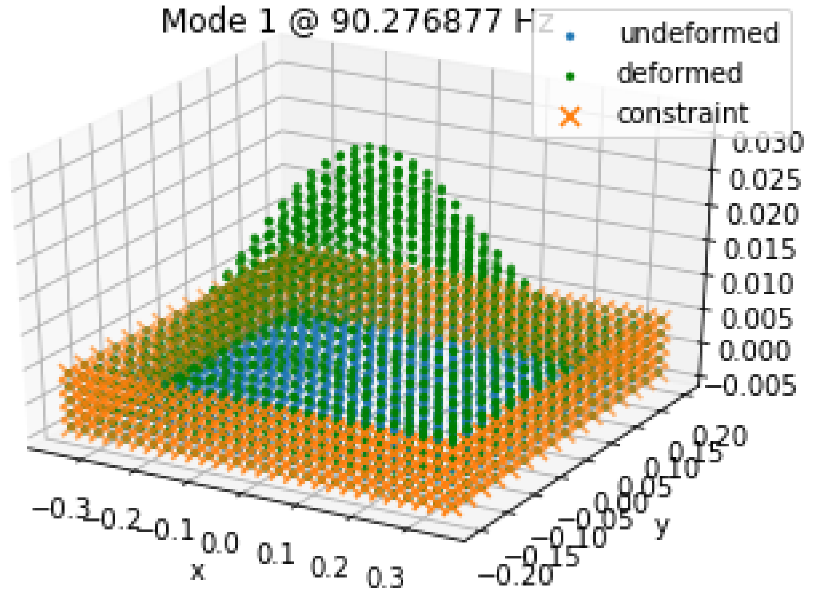


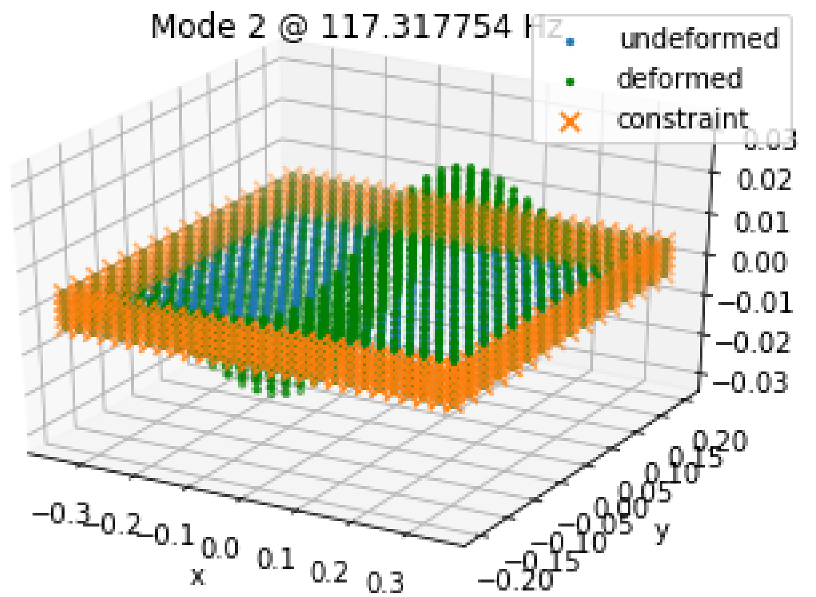
Mode 9 @ 383.695469 Hz

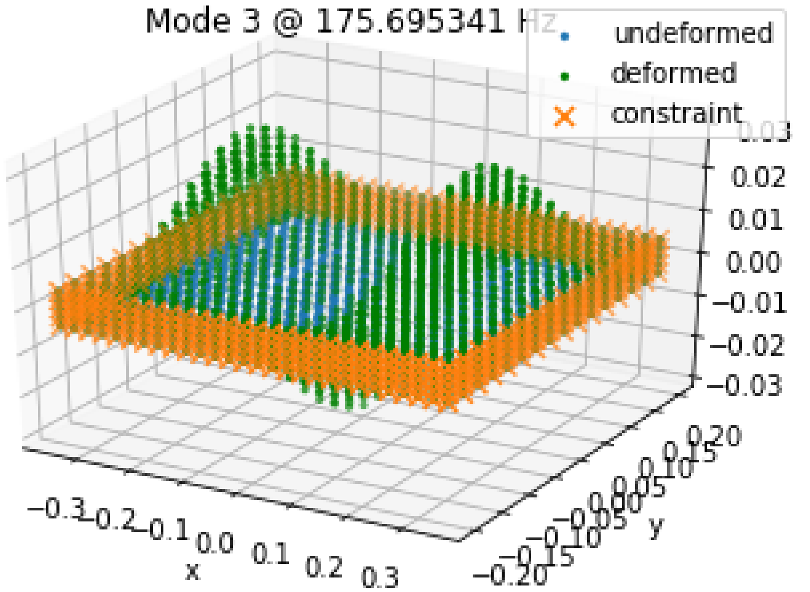


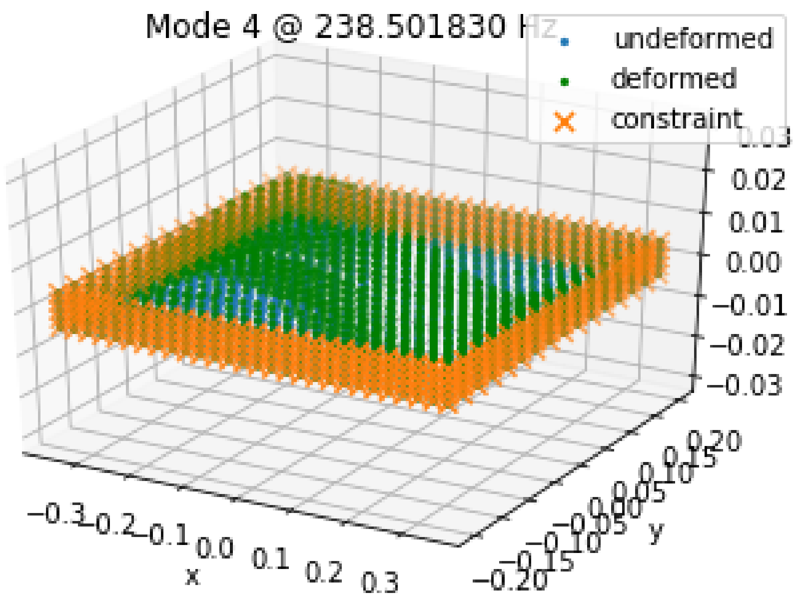
Mode 10 @ 460.570309 Hz

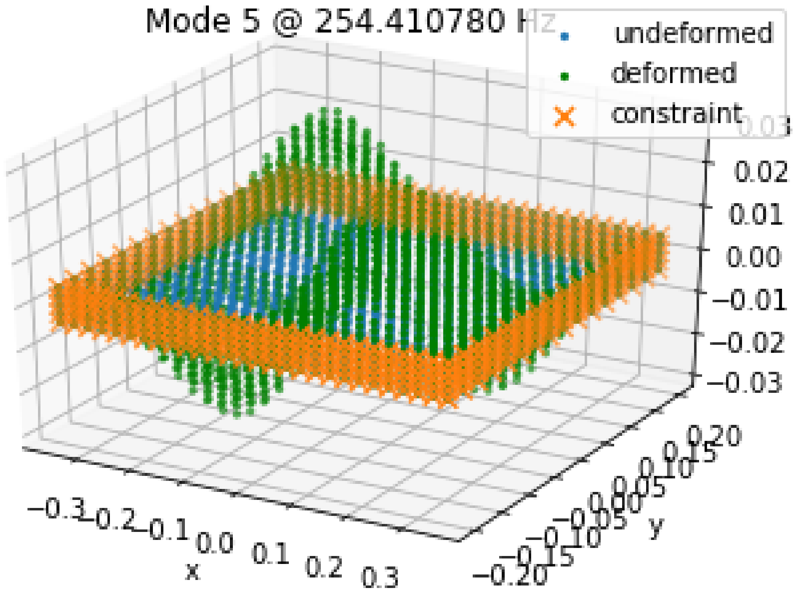


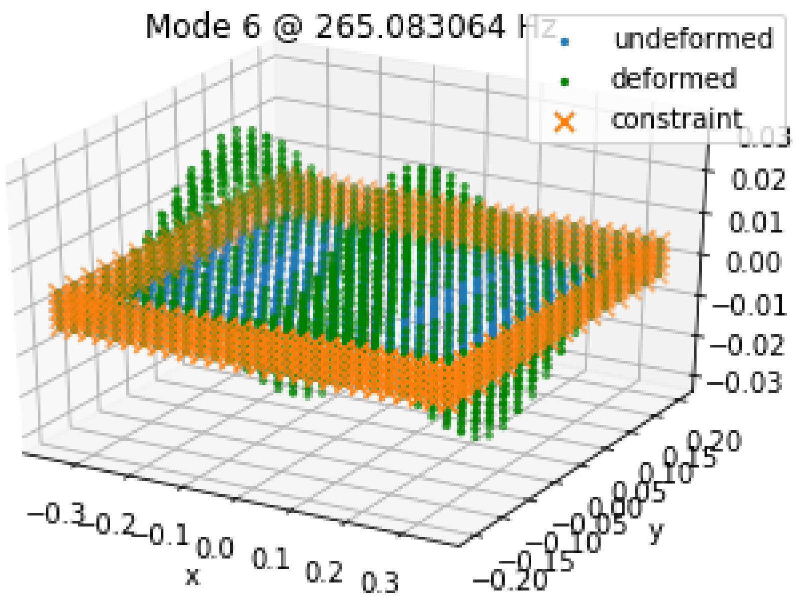

Mode 1 @ 90.276877 Hz
undeformed
deformed
constraint
0.030
0.025
0.020
0.015
0.010
0.005
0.000
−0.005
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
y

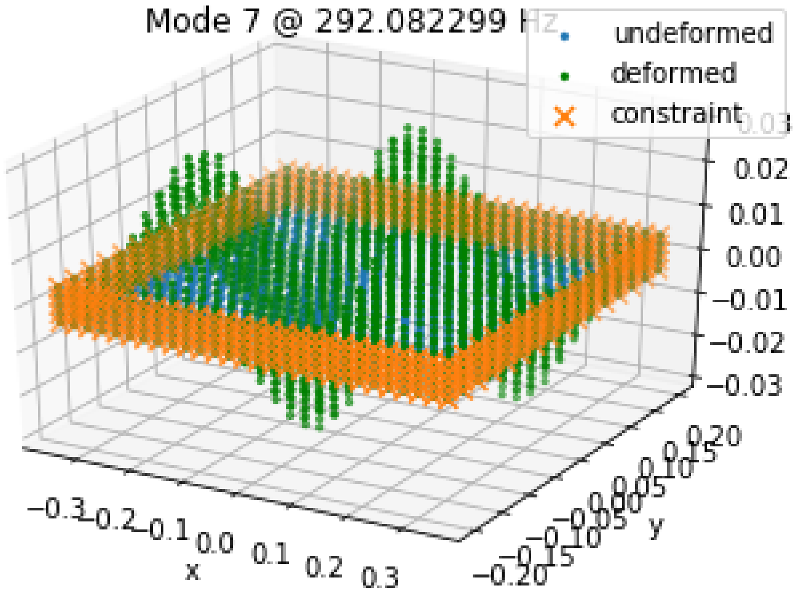

Mode 2 @ 117.317754 Hz
undeformed
deformed
constraint
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
y

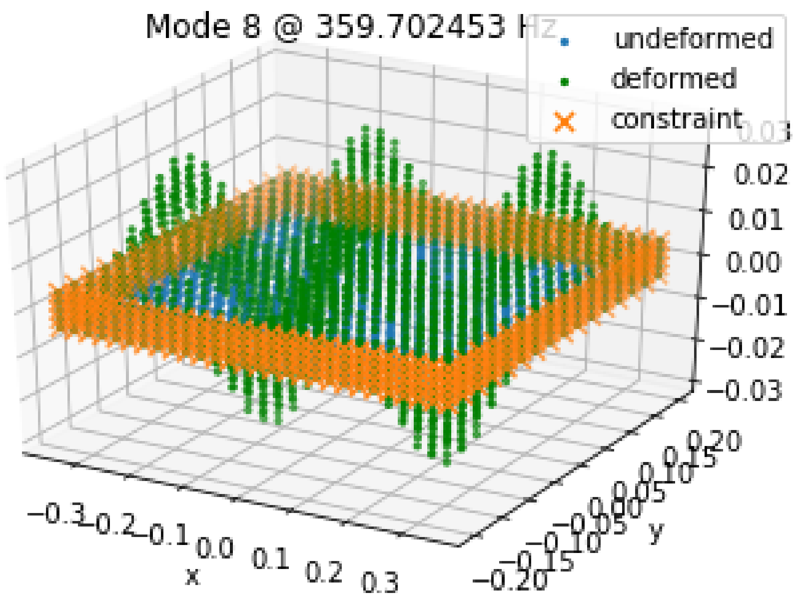

Mode 3 @ 175.695341 Hz
undeformed
deformed
constraint
x
y
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20

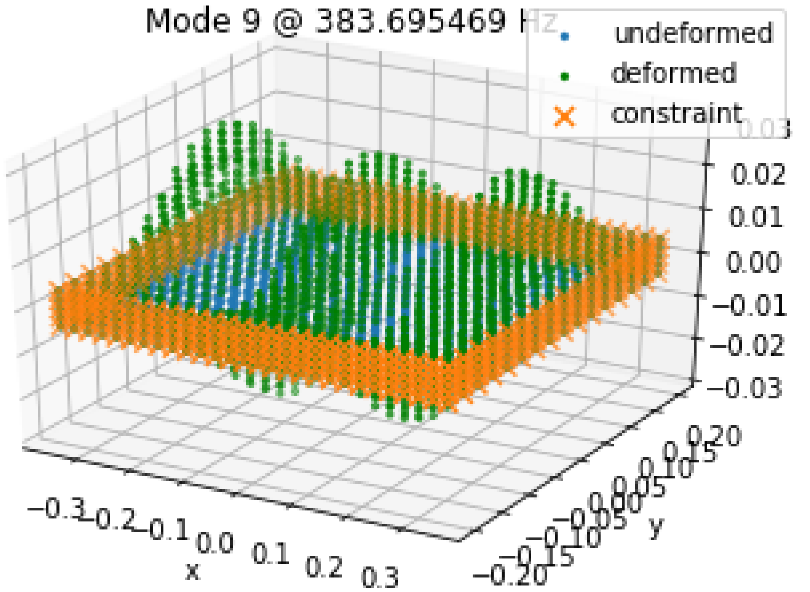

Mode 4 @ 238.501830 Hz
undeformed
deformed
constraint
x
y
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20

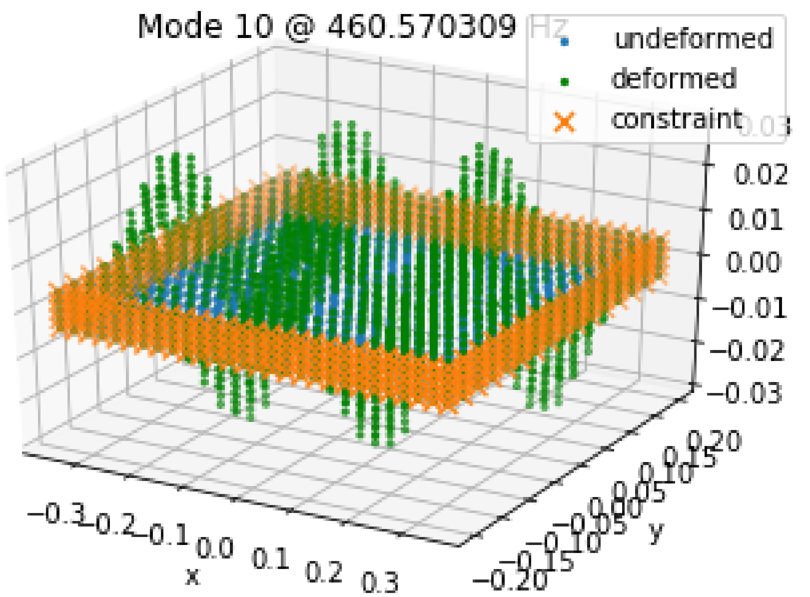

Mode 5 @ 254.410780 Hz
undeformed
deformed
constraint
x
y


Mode 6 @ 265.083064 Hz
undeformed
deformed
constraint
x
y


Mode 7 @ 292.082299 Hz
undeformed
deformed
constraint
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
y


Mode 8 @ 359.702453 Hz
undeformed
deformed
constraint
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
y





4.1 Aminations

You can use the excellent JSanimation package to show matplotlib animations in a jupyter notebook.

[10]:

```

```

```
Nt = np.argwhere(np.abs(X[:,2]-X[:,2].max())<tol).ravel() # Node indices of the␣
↪top layer nodes
Nb = np.argwhere(np.abs(X[:,2]-X[:,2].min())<tol).ravel() # Node indices of the␣
↪bottom layer nodes
```

[11]:
```
import matplotlib.animation as animation

def giveMeAnimation(i,whatTOdo):
    plt.rcParams["animation.html"] = "jshtml"

    u = np.zeros(3*N) # initialize displacement vector
    uc = np.real(V[:,i]) #without exp. power term, since we only look at the␣
↪static displacment
    u[~Ic] = uc

    # format U like X
    U = np.array([u[Ix],u[Iy],u[Iz]]).T

    #-------------------------------------------------------------
    # set up figure and animation
    fig, ax = plt.subplots(subplot_kw={'projection':'3d'})
    x, y, z = [],[],[]
    sc = ax.scatter(x,y,z,s=5,label='Mode_' + str(i))
    ax.set_xlim(-0.3,0.3)
    ax.set_ylim(-0.2,0.2)
    ax.set_zlim(-0.04,0.04)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    ax.legend()

    def init():
        """initialize animation"""
        sc._offsets3d = ([],[],[])
        return sc

    def animate(i):
        # scale factor for plotting
        s = 0.5/np.max(np.sqrt(np.sum(U**2,axis=0)))*np.sin(1/25*2*np.pi*i)
        Xu = X + s*U # defomed configuration (displacement scaled by s)

        x = np.ndarray.tolist(Xu[:,0])
        y = np.ndarray.tolist(Xu[:,1])
        z = np.ndarray.tolist(Xu[:,2])

        sc._offsets3d = (x,y,z)
        return sc
```

```
    ani = animation.FuncAnimation(fig, animate, frames=50,
                                  interval=100, init_func=init)

    # # save the animation as an mp4.  This requires ffmpeg or mencoder to be
    # # installed.  The extra_args ensure that the x264 codec is used, so that
    # # the video can be embedded in html5.  You may need to adjust this for
    # # your system: for more information, see
    # # http://matplotlib.sourceforge.net/api/animation_api.html

    if whatTOdo == 'Save' :
        ani.save('modalanalyse_mode_' + str(i) + '.mp4', fps=30,␣
↪extra_args=['-vcodec', 'libx264'])
    elif whatTOdo == 'justShow':
        plt.close()
        return ani

def giveMeAnimation_fancy(i,whatTOdo):
    plt.rcParams["animation.html"] = "jshtml"

    u = np.zeros(3*N) # initialize displacement vector
    uc = np.real(V[:,i]) #without exp. power term, since we only look at the␣
↪static displacment
    u[~Ic] = uc

    # format U like X
    U = np.array([u[Ix],u[Iy],u[Iz]]).T

    #------------------------------------------------------------
    # set up figure and animation
    fig, ax = plt.subplots(subplot_kw={'projection':'3d'})

    # Plot a basic wireframe.
    index = 1

    x_bot = np.reshape(X[Nb,0],(len(y),len(x)))
    y_bot = np.reshape(X[Nb,1],(len(y),len(x)))
    z_bot = np.reshape(X[Nb,2],(len(y),len(x)))

    x_top = np.reshape(X[Nt,0],(len(y),len(x)))
    y_top = np.reshape(X[Nt,1],(len(y),len(x)))
    z_top = np.reshape(X[Nt,2],(len(y),len(x)))

    x_o = np.reshape(X[No,0],(len(y),len(z)))
    y_o = np.reshape(X[No,1],(len(y),len(z)))
    z_o = np.reshape(X[No,2],(len(y),len(z)))
```

```
x_n = np.reshape(X[Nn,0],(len(x),len(z)))
y_n = np.reshape(X[Nn,1],(len(x),len(z)))
z_n = np.reshape(X[Nn,2],(len(x),len(z)))

x_s = np.reshape(X[Ns,0],(len(x),len(z)))
y_s = np.reshape(X[Ns,1],(len(x),len(z)))
z_s = np.reshape(X[Ns,2],(len(x),len(z)))

x_w = np.reshape(X[Nw,0],(len(y),len(z)))
y_w = np.reshape(X[Nw,1],(len(y),len(z)))
z_w = np.reshape(X[Nw,2],(len(y),len(z)))

sf1 = ax.plot_surface(x_bot, y_bot, z_bot, rstride=index, cstride=index)
sf2 = ax.plot_surface(x_top, y_top, z_top, rstride=index, cstride=index)
sf3 = ax.plot_surface(x_o, y_o, z_o, rstride=index, cstride=index)
sf4 = ax.plot_surface(x_n, y_n, z_n, rstride=index, cstride=index)
sf5 = ax.plot_surface(x_s, y_s, z_s, rstride=index, cstride=index)
sf6 = ax.plot_surface(x_w, y_w, z_w, rstride=index, cstride=index)

ax.set_xlim(-0.3,0.3)
ax.set_ylim(-0.2,0.2)
ax.set_zlim(-0.04,0.04)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

def animate(i):
    # scale factor for plotting
    s = 0.5/np.max(np.sqrt(np.sum(U**2,axis=0)))*np.sin(1/25*2*np.pi*i)
    Xu = X + s*U # defomed configuration (displacement scaled by s)

    x_bot = np.reshape(Xu[Nb,0],(len(y),len(x)))
    y_bot = np.reshape(Xu[Nb,1],(len(y),len(x)))
    z_bot = np.reshape(Xu[Nb,2],(len(y),len(x)))

    x_top = np.reshape(Xu[Nt,0],(len(y),len(x)))
    y_top = np.reshape(Xu[Nt,1],(len(y),len(x)))
    z_top = np.reshape(Xu[Nt,2],(len(y),len(x)))

    x_o = np.reshape(Xu[No,0],(len(y),len(z)))
    y_o = np.reshape(Xu[No,1],(len(y),len(z)))
    z_o = np.reshape(Xu[No,2],(len(y),len(z)))

    x_n = np.reshape(Xu[Nn,0],(len(x),len(z)))
    y_n = np.reshape(Xu[Nn,1],(len(x),len(z)))
    z_n = np.reshape(Xu[Nn,2],(len(x),len(z)))
```

```
        x_s = np.reshape(Xu[Ns,0],(len(x),len(z)))
        y_s = np.reshape(Xu[Ns,1],(len(x),len(z)))
        z_s = np.reshape(Xu[Ns,2],(len(x),len(z)))

        x_w = np.reshape(Xu[Nw,0],(len(y),len(z)))
        y_w = np.reshape(Xu[Nw,1],(len(y),len(z)))
        z_w = np.reshape(Xu[Nw,2],(len(y),len(z)))

        ax.clear()
        sf1 = ax.plot_surface(x_bot,y_bot,z_bot,rstride=index, cstride=index)
        sf2 = ax.plot_surface(x_top,y_top,z_top,rstride=index, cstride=index)
        sf3 = ax.plot_surface(x_o,y_o,z_o,rstride=index, cstride=index)
        sf4 = ax.plot_surface(x_n,y_n,z_n,rstride=index, cstride=index)
        sf5 = ax.plot_surface(x_s,y_s,z_s,rstride=index, cstride=index)
        sf6 = ax.plot_surface(x_w,y_w,z_w,rstride=index, cstride=index)

        ax.set_xlim(-0.3,0.3)
        ax.set_ylim(-0.2,0.2)
        ax.set_zlim(-0.04,0.04)
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')

        return sf1,sf2,sf3,sf5,sf6,sf4

    ani = animation.FuncAnimation(fig, animate, frames=50, interval=100)

    # # save the animation as an mp4.  This requires ffmpeg or mencoder to be
    # # installed.  The extra_args ensure that the x264 codec is used, so that
    # # the video can be embedded in html5.  You may need to adjust this for
    # # your system: for more information, see
    # # http://matplotlib.sourceforge.net/api/animation_api.html

    if whatTOdo == 'Save' :
        ani.save('modalanalyse_mode_' + str(i) + '.mp4', fps=30,␣
↪extra_args=['-vcodec', 'libx264'])
    elif whatTOdo == 'justShow':
        plt.close()
        return ani
```

[13]:
```
giveMeAnimation_fancy(0,'justShow')
```

[13]: <matplotlib.animation.FuncAnimation at 0x280005bd648>

5 Determine useful damping

The damping ratio for each mode is computed as follows for Rayleigh damping

$$\zeta = \frac{\alpha}{2\omega} + \frac{\beta\omega}{2}$$

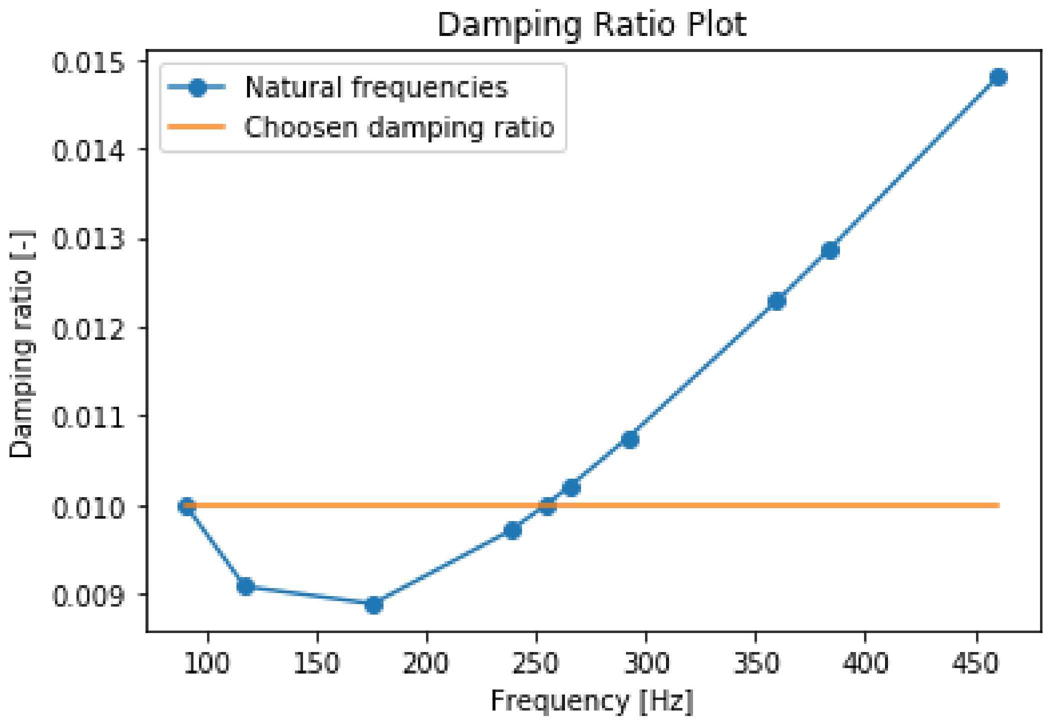
Defining two frequencies ω_1 and ω_2 and corresponding damping ratios we can compute α and β.

Plot the damping ratio in the frequency range of the first 10 natural frequencies. Choose alpha and beta such that the damping ratio is = 0.01 for mode 1 and mode 5.

[14]:
```
## Compute
omegas = np.sqrt(abs(W)) # Collect angular eigenfrq.
omegaCoeffs = np.vstack((1/omegas, omegas)).T # Build coefficent matrix

dampingRatio = 0.01 # Damping ratio choosen
b = dampingRatio*np.ones(np.shape(omegaCoeffs)[0]) # Right-hand side of␣
↪omegaCoeffs*alphaBeta = b

alphaBeta = np.linalg.solve(omegaCoeffs[(0,4),:], b.take([0,4])) # Solve for␣
↪alphaBeta at 1. and 5. natural frequency

dampingRatios = omegaCoeffs @ alphaBeta
```

[15]:
```
## Plot
fig, ax = plt.subplots()  # Create a figure and an axes.
ax.plot(omegas/2/np.pi, dampingRatios, '-o', label="Natural frequencies")  #␣
↪Plot damping ratio.
ax.plot(omegas/2/np.pi, np.ones_like(dampingRatios)*dampingRatio,␣
↪label="Choosen damping ratio") # Plot choosen damping ratio.
ax.set_xlabel('Frequency [Hz]')  # Add an x-label to the axes.
ax.set_ylabel('Damping ratio [-]')  # Add a y-label to the axes.
ax.set_title("Damping Ratio Plot")  # Add a title to the axes.
ax.legend()  # Add a legend.
print(f"alpha={alphaBeta[0] :.3e} and beta={alphaBeta[1] :.3e}")
```

```
alpha=4.187e+00 and beta=4.617e-06
```



6 Time domain

First we investigate the system in time domain. It should be loaded by a transient force in z-direction at point P1.

6.1 Excitation signal

Use a smooth-step or smooth-impule function with suitable time constant. As suitable time constant will excite interesting dynamics in the system.

The period of the first natural frequency can act as a guideline for the time constant. If the transient excitation is very slow (it takes longer than the period of the lowest eigenfrequency to reach its maximum) there will be no significant dynamics. If the transient excitation is very fast (pulses with frequency content covering many natural frequencies) it may excite significant dynamics.

Experiment with different excitation signals. Plot the signal over time, and Fourier transform it to show its frequency content.

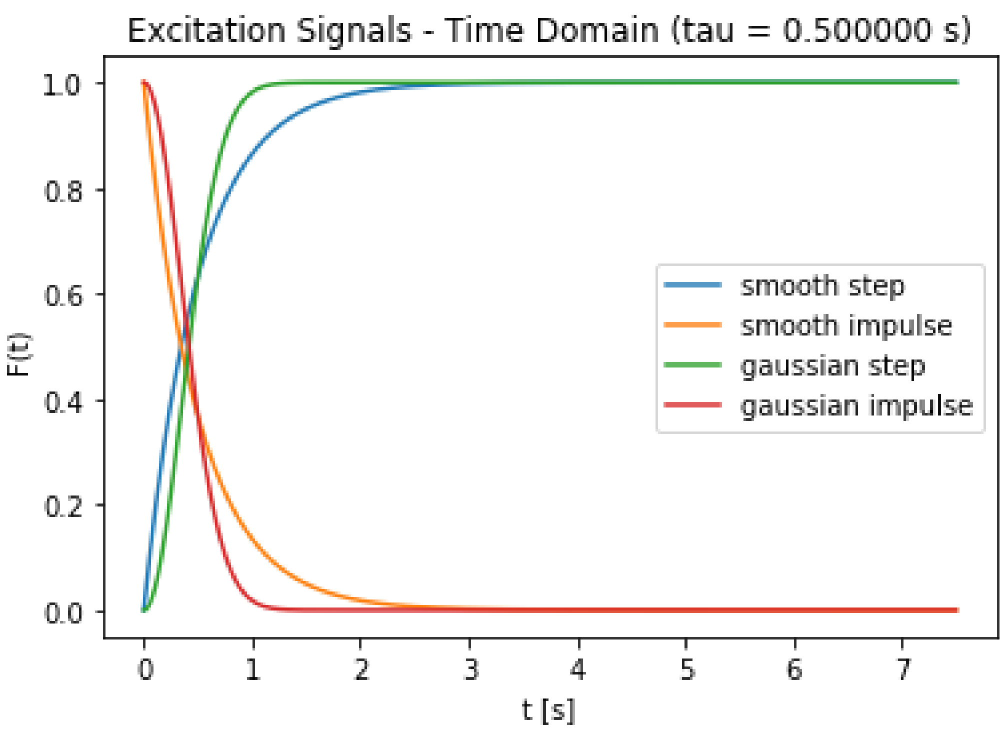
```
from numpy.fft import rfft, rfftfreq
```

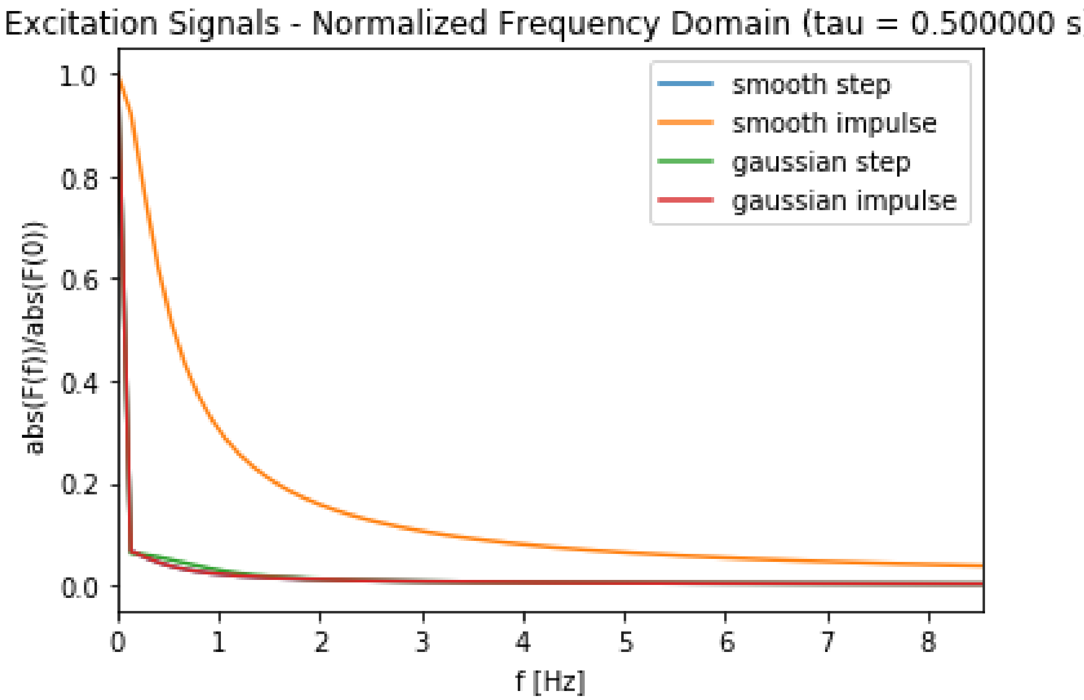
[16]:
```
## Define some excitation signals

def smoothImpulse(t, tau=1, t0=0):
    return np.exp(-(t-t0)/tau)
```

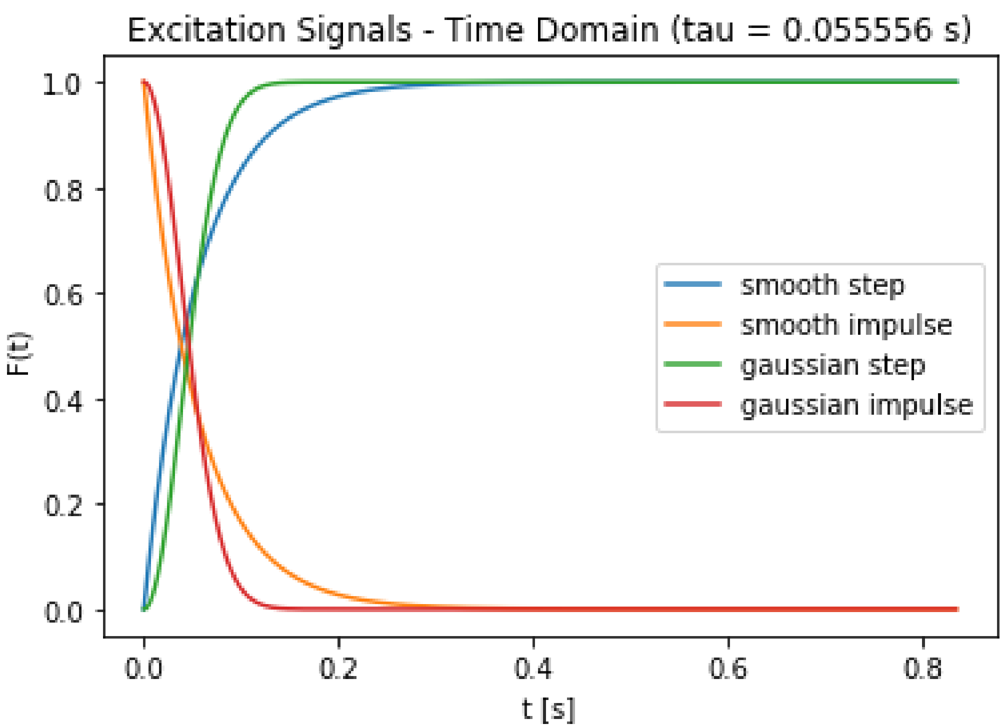
```
def smoothStep(t, tau=1, t0=0):
    return 1-smoothImpulse(t, tau, t0)

def gaussianImpulse(t, tau=1, t0=0):
    return np.exp(-((t-t0)/tau)**2)

def gaussianStep(t, tau=1, t0=0):
    return 1-gaussianImpulse(t, tau, t0)
```

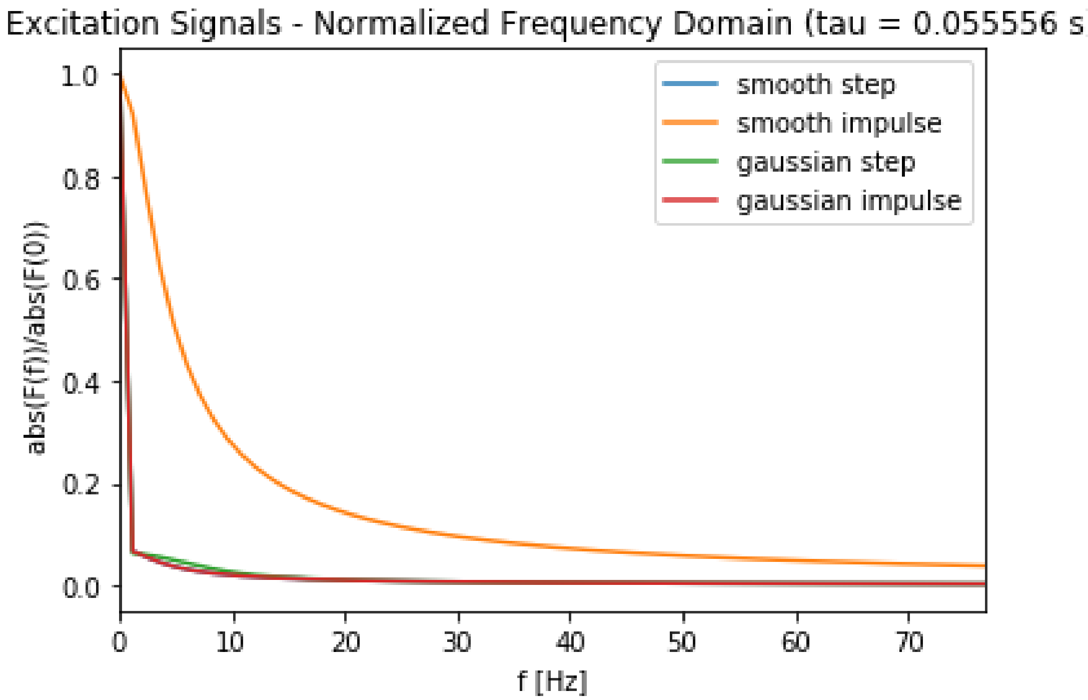
[17]:
```
## Plot excitation signals

for tau in [0.5, 5/90, 1/90, 0.002]: # Iterate over a few manually defined time␣
↪constants

    T = 15*tau  # Time duration of signal in sec
    N = 2**9  # Size of sample array
    t = np.linspace(0, T, N)  # Time array of size N
    Fs = N/T  # Sampling frequency
    f = np.linspace(0, Fs/2, N//2 + 1)  # One sided positive frequency array

    # Time domain
    timePlot, timeAxis = plt.subplots()

    timeAxis.plot(t, smoothStep(t, tau), label = "smooth step")
    timeAxis.plot(t, smoothImpulse(t, tau), label = "smooth impulse")
    timeAxis.plot(t, gaussianStep(t, tau), label = "gaussian step")
    timeAxis.plot(t, gaussianImpulse(t, tau), label = "gaussian impulse")
    timeAxis.set_xlabel('t [s]')
    timeAxis.set_ylabel('F(t)')
    timeAxis.set_title(f"Excitation Signals - Time Domain (tau = {tau:3f} s)")
    timeAxis.legend()

    # Frequency domain
    frqPlot, frqAxis = plt.subplots()

    frqAxis.plot(f,
                 np.abs(rfft(smoothStep(t, tau)))/np.abs(rfft(smoothStep(t,␣
↪tau)))[0],
                 label = "smooth step")
    frqAxis.plot(f,
                 np.abs(rfft(smoothImpulse(t, tau)))/np.
↪abs(rfft(smoothImpulse(t, tau)))[0],
                 label = "smooth impulse")
    frqAxis.plot(f,
                 np.abs(rfft(gaussianStep(t, tau)))/np.abs(rfft(gaussianStep(t,␣
↪tau)))[0],
                 label = "gaussian step")
```

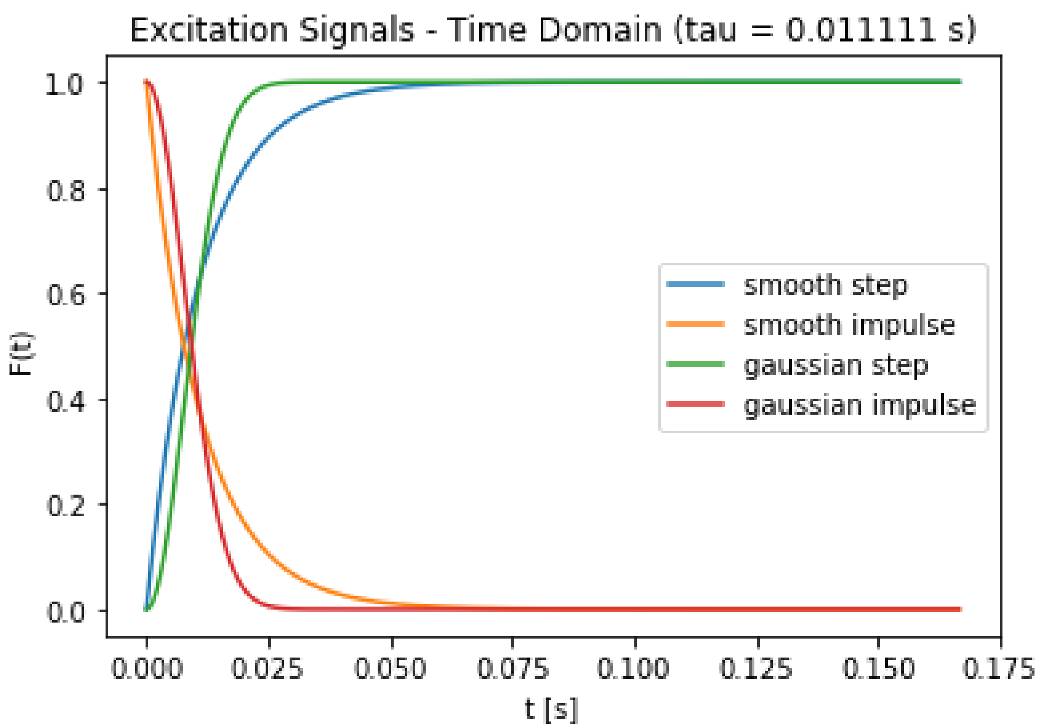
```
    frqAxis.plot(f,
                 np.abs(rfft(smoothStep(t, tau)))/np.abs(rfft(smoothStep(t,␣
↪tau)))[0],
                 label = "gaussian impulse")

    frqAxis.set_xlabel('f [Hz]')
    frqAxis.set_ylabel('abs(F(f))/abs(F(0)) ')
    frqAxis.set_title(f"Excitation Signals - Normalized Frequency Domain (tau =␣
↪{tau:3f} s)")
    frqAxis.set_xlim(0,f[-1]/4)

    frqAxis.legend()
```

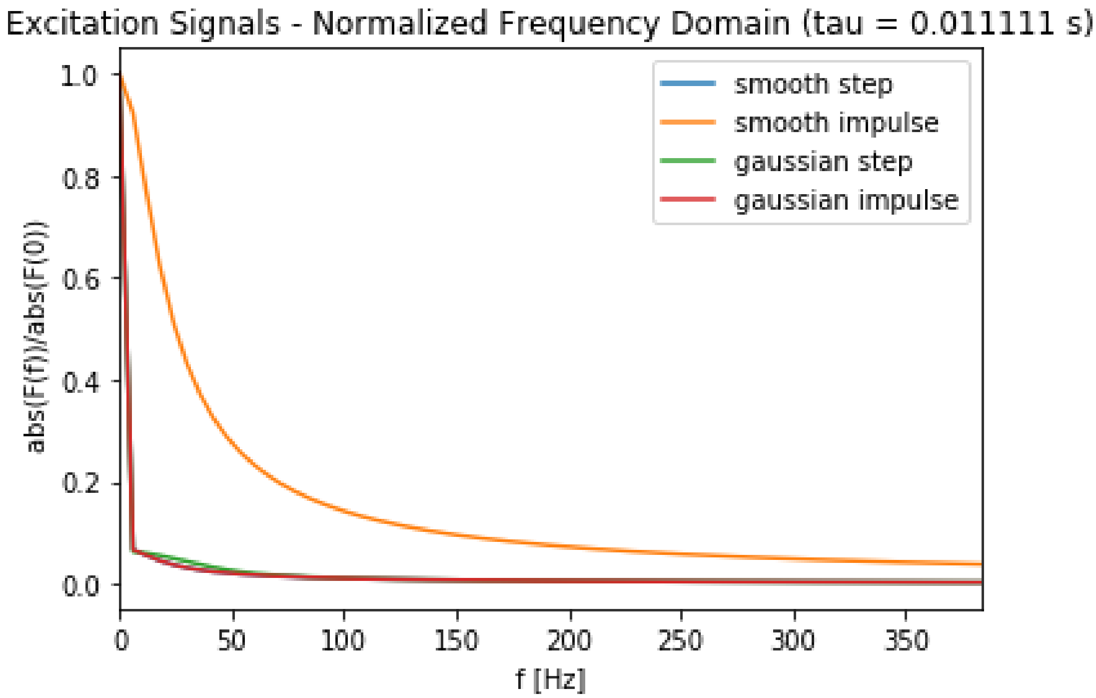


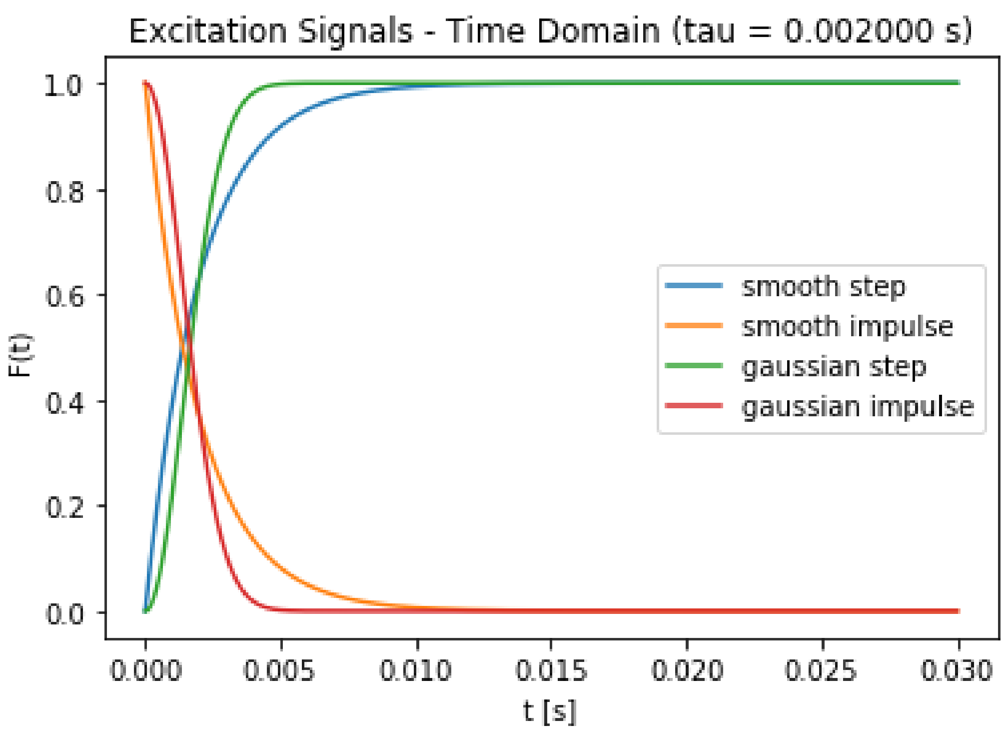

Excitation Signals - Normalized Frequency Domain (tau = 0.500000 s)
abs(F(f))/abs(F(0))
f [Hz]
smooth step
smooth impulse
gaussian step
gaussian impulse

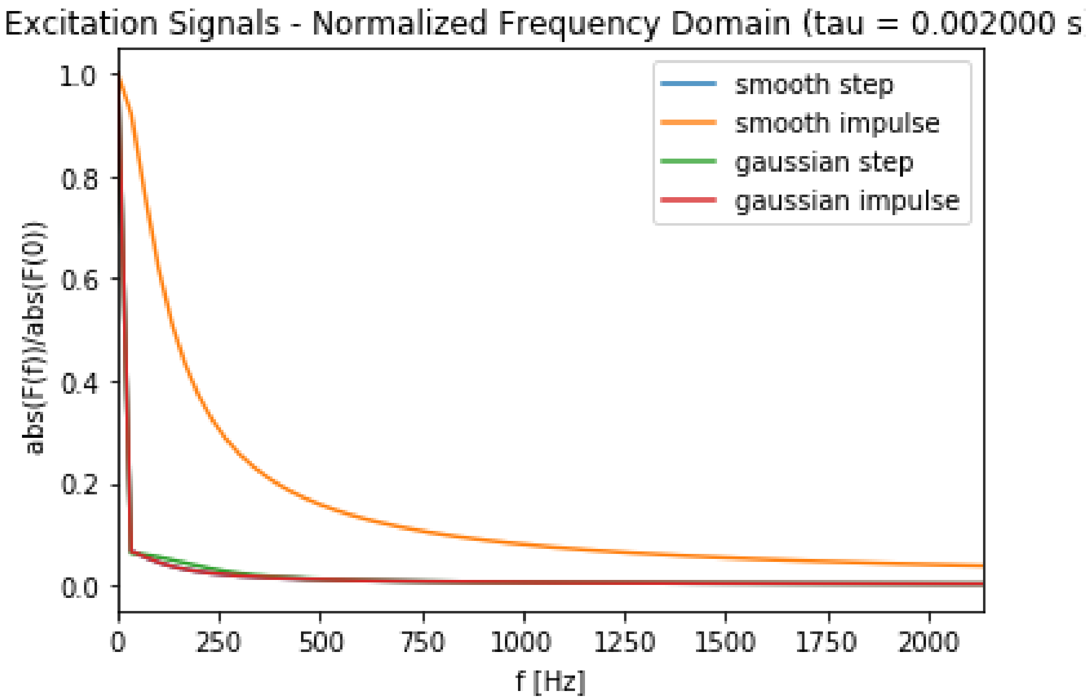

Excitation Signals - Time Domain (tau = 0.055556 s)
F(t)
t [s]
smooth step
smooth impulse
gaussian step
gaussian impulse


Excitation Signals - Normalized Frequency Domain (tau = 0.055556 s)
abs(F(f))/abs(F(0))
f [Hz]
smooth step
smooth impulse
gaussian step
gaussian impulse
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60
70


Excitation Signals - Time Domain (tau = 0.011111 s)
F(t)
t [s]
smooth step
smooth impulse
gaussian step
gaussian impulse
0.0
0.2
0.4
0.6
0.8
1.0
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175


Excitation Signals - Normalized Frequency Domain (tau = 0.011111 s)
abs(F(f))/abs(F(0))
f [Hz]
smooth step
smooth impulse
gaussian step
gaussian impulse
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
350


Excitation Signals - Time Domain (tau = 0.002000 s)
F(t)
t [s]
smooth step
smooth impulse
gaussian step
gaussian impulse
1.0
0.8
0.6
0.4
0.2
0.0
0.000
0.005
0.010
0.015
0.020
0.025
0.030



6.2 Task 1: Compute the transient response

Assume a load $f(t) = 1 - e^{-(t/0.002)^2}$ in z-direction at P1. Compute the response of the plate for $0 < t < 0.2$ and plot the time evolution of the z-displacement at the center of the plate, at P1 and at P2.

Estimate the oscillation frequency of the system from the time signal. How many frequencies do your see in the signal for the center point, how many in P1 and P2?

6.2.1 Note:

For TASK 1 the damped system is investigated. The damping paramters α and β follow from

$$\zeta(\omega_{n1,5}) := 0.01,$$

and the proportional damping matrix is computed via

$$\mathbf{C} = \alpha\mathbf{M} + \beta\mathbf{K}.$$

[31]:
```
## Functions for TASK 1:

# Define some excitation signals (again for completeness)

def smoothImpulse(t, tau=1, t0=0):
    return np.exp(-(t-t0)/tau)
```

```
def smoothStep(t, tau=1, t0=0):
    return 1-smoothImpulse(t, tau, t0)

def gaussianImpulse(t, tau=1, t0=0):
    return np.exp(-((t-t0)/tau)**2)

def gaussianStep(t, tau=1, t0=0): # <-- Thats the one for Task 1 !
    return 1-gaussianImpulse(t, tau, t0)

def unreduce_constrained(uc, Ic):
    """Takes the reduced displacement array uc of shape(m,) and the boolean array Ic of shape(n,)
    and builds a new unreduced u array of shape(n,)."""
    u = np.zeros((Ic.shape[0], uc.shape[1])) # Initialize unconstrained displacement array
    u[~Ic] = uc
    return u

def animate_plate_timedomain(u, time, X,
                             scaling_factor=2500,
                             exportFile=False,
                             fileName="timedomain-animation.mp4"):
    """This function takes a displacement array u and animates the time evolution of
    a plate with clamped edges. """

    # Setting constants
    MAX_FRAMES = 20 # Frames are evenly spaced out. Higher value means longer computation and better resolution.
    FRAME_COUNTER_POSITION = (.025, .975) # Top-left in XY coordinates
    TIME_COUNTER_POSITION = (.6, .975) # Top-right-ish
    FPS_EXPORT = 10

    # Obscure setting for plt to display animation correctly.
    plt.rcParams["animation.html"] = "jshtml"

    # Set up figure and animation
    fig, ax = plt.subplots(subplot_kw={'projection': '3d'})


    def animation_callback_function(i):

        ut = u[:, i]  # Assign displacements at frame/timestep.
        Ut = np.array([ut[Ix], ut[Iy], ut[Iz]]).T  # format ut into column oriented array Ut [ut_x, ut_y, ut_z]
        Xt = X + scaling_factor * Ut  # Position Xt of nodes at frame = inital position X0 + displacement at frame.
```

```python
    # Assign faces coordinates
    x_bot = np.reshape(Xt[Nb, 0], (len(y), len(x)))
    y_bot = np.reshape(Xt[Nb, 1], (len(y), len(x)))
    z_bot = np.reshape(Xt[Nb, 2], (len(y), len(x)))

    x_top = np.reshape(Xt[Nt, 0], (len(y), len(x)))
    y_top = np.reshape(Xt[Nt, 1], (len(y), len(x)))
    z_top = np.reshape(Xt[Nt, 2], (len(y), len(x)))

    x_o = np.reshape(Xt[No, 0], (len(y), len(z)))
    y_o = np.reshape(Xt[No, 1], (len(y), len(z)))
    z_o = np.reshape(Xt[No, 2], (len(y), len(z)))

    x_n = np.reshape(Xt[Nn, 0], (len(x), len(z)))
    y_n = np.reshape(Xt[Nn, 1], (len(x), len(z)))
    z_n = np.reshape(Xt[Nn, 2], (len(x), len(z)))

    x_s = np.reshape(Xt[Ns, 0], (len(x), len(z)))
    y_s = np.reshape(Xt[Ns, 1], (len(x), len(z)))
    z_s = np.reshape(Xt[Ns, 2], (len(x), len(z)))

    x_w = np.reshape(Xt[Nw, 0], (len(y), len(z)))
    y_w = np.reshape(Xt[Nw, 1], (len(y), len(z)))
    z_w = np.reshape(Xt[Nw, 2], (len(y), len(z)))

    ax.clear() # Clear axis object from last call.

    index = 1

    # Draw faces: bottom, top, ...
    sf1 = ax.plot_surface(x_bot, y_bot, z_bot, rstride=index, cstride=index)
    sf2 = ax.plot_surface(x_top, y_top, z_top, rstride=index, cstride=index)
    sf3 = ax.plot_surface(x_o, y_o, z_o, rstride=index, cstride=index)
    sf4 = ax.plot_surface(x_n, y_n, z_n, rstride=index, cstride=index)
    sf5 = ax.plot_surface(x_s, y_s, z_s, rstride=index, cstride=index)
    sf6 = ax.plot_surface(x_w, y_w, z_w, rstride=index, cstride=index)

    # Note: there's certainly are more general way to scale the axis␣
↪properly!
    ax.set_xlim(-0.3, 0.3)
    ax.set_ylim(-0.2, 0.2)
    ax.set_zlim(-0.04, 0.04)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
```

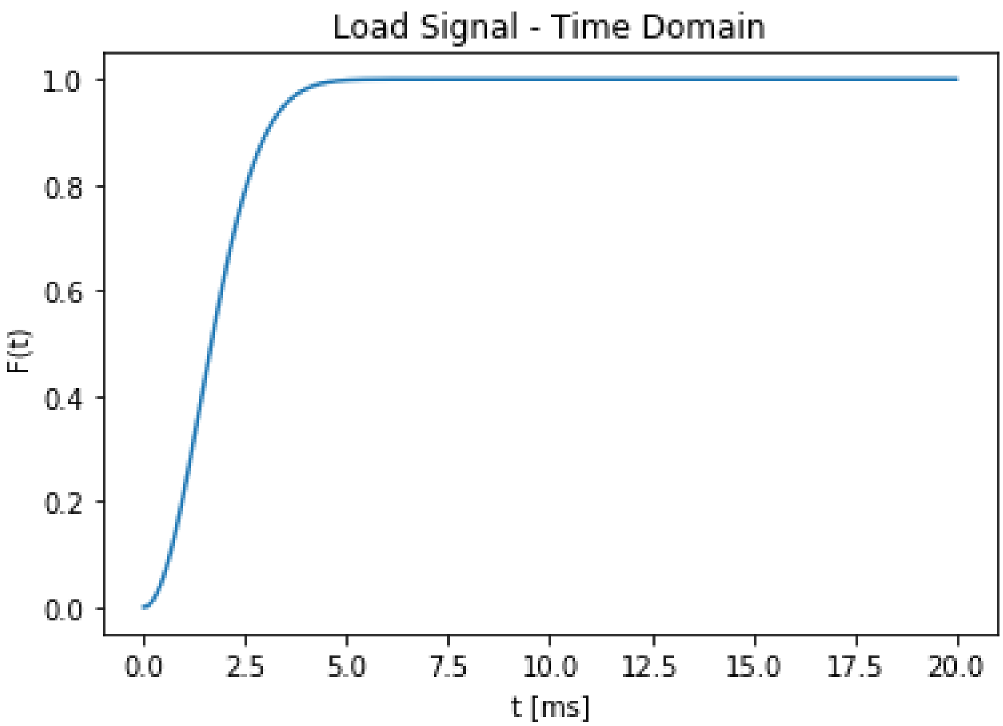
```
        # Display and update frame counter on the plot
        frame_count = i*MAX_FRAMES//np.shape(time)[0]
        frame_info = f"Frame: {frame_count}" # For Python >=3.6, f-strings make␣
↪string literals much more readable :)!
        ax.annotate(frame_info,
                    xy=FRAME_COUNTER_POSITION, xycoords='figure fraction',
                    horizontalalignment='left', verticalalignment='top',
                    fontsize=15)

        # Display and update real-time counter on the plot
        dt = time[1] - time[0] # Timestep in seconds
        current_time = i*dt
        time_info = f"t = {current_time*1000 :.5f} ms"
        ax.annotate(time_info,
                    xy=TIME_COUNTER_POSITION, xycoords='figure fraction',
                    horizontalalignment='left', verticalalignment='top',
                    fontsize=15)


        return sf1, sf2, sf3, sf5, sf6, sf4

    # Set animation object
    ani = animation.FuncAnimation(fig,
                                  animation_callback_function,
                                  frames=range(0, np.shape(time)[0], np.
↪shape(time)[0]//MAX_FRAMES),
                                  blit=True)

    # # save the animation as an mp4.  This requires ffmpeg or mencoder to be
    # # installed.  The extra_args ensure that the x264 codec is used, so that
    # # the video can be embedded in html5.  You may need to adjust this for
    # # your system: for more information, see
    # # http://matplotlib.sourceforge.net/api/animation_api.html

    if exportFile:
        ani.save(fileName, fps=FPS_EXPORT, extra_args=['-vcodec', 'libx264'])
    else:
        plt.close()
        return ani

def plot_load_analysis(load, t):
    """This function plots a given load function array in the time and␣
↪frequency domain."""

    # Time domain
    timePlot, timeAxis = plt.subplots()
    timeAxis.plot(t*1000, load, label = "load signal")
```

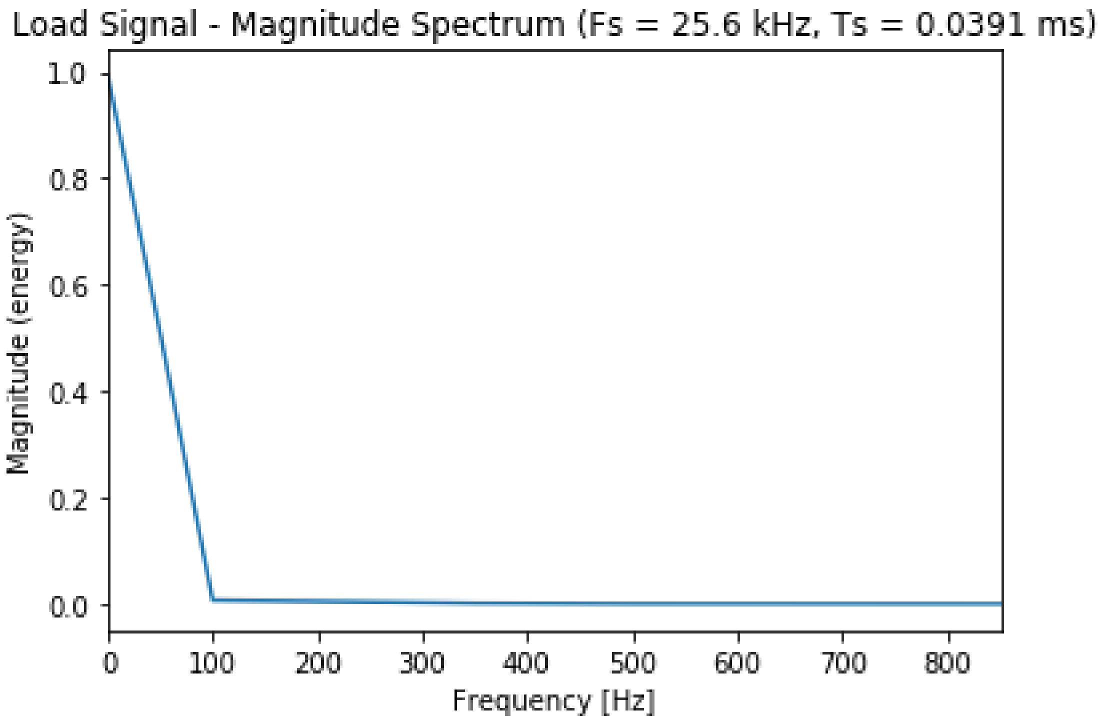
```
    timeAxis.set_xlabel('t [ms]')
    timeAxis.set_ylabel('F(t)')
    timeAxis.set_title(f"Load Signal - Time Domain")


    # Frequency domain
    Fs = 1 / (t[1] - t[0])
    frqPlot, frqAxis = plt.subplots()
    frqAxis.magnitude_spectrum(load, Fs, scale='linear')
    frqAxis.set_xlim(0,Fs/30) # Note: If time left implement smart scaling.
    frqAxis.set_xlabel('Frequency [Hz]')
    frqAxis.set_title(f"Load Signal - Magnitude Spectrum (Fs = {Fs/1000 :.3}␣
 ↪kHz, Ts = {1/Fs*1000 :.3} ms)")

def plot_displacements_timedomain(u, time, N1, N2):

    # Find node number in the center
    P_center = [0.,0.,0.]
    N_center = np.argmin(np.sum((X-P_center)**2,axis=1))

    # Plot
    timePlot, timeAxis = plt.subplots()
    timeAxis.plot(time*1000, u[N1], label = "P1")
    timeAxis.plot(time*1000, u[N2], label = "P2")
    timeAxis.set_xlabel('t [ms]')
    timeAxis.set_ylabel('u(t) [m]')
    timeAxis.legend()

    timePlot_center, timeAxis_center = plt.subplots()
    timeAxis_center.plot(time*1000, u[N_center], label = "P_center")
    timeAxis_center.set_xlabel('t [ms]')
    timeAxis_center.set_ylabel('u(t) [m]')
    timeAxis_center.set_title(f"Displacements - Time Domain")
    timeAxis_center.legend()

def plot_P1_timedomain(u, time, N1):

    # Plot
    timePlot, timeAxis = plt.subplots()
    timeAxis.plot(time*1000, u[N1], label = "P1")
    timeAxis.set_xlabel('t [ms]')
    timeAxis.set_ylabel('u(t) [m]')
    timeAxis.set_title(f"Displacement - Time Domain")
    timeAxis.legend()

def full_excitation_analysis(tau=0.002, T=0.2,
                             excitation_type='step',
```

```python
                              display_animation=False):

    # Assign load
    if excitation_type == 'step':

        # integration time
        f_max = 1/tau # Very crude estimation of max frequency.
        dt = 1/(20*f_max) # Timestep
        time = np.arange(0, T, dt) # Create time array for integration

        load = gaussianStep(time, tau)

    elif excitation_type == 'impulse':

        # integration time
        f_max = 5/tau # Very crude estimation of max frequency.
        dt = 1/(20*f_max) # Timestep
        time = np.arange(0, T, dt) # Create time array for integration

        load = gaussianImpulse(time, tau)

    # Checkout load function
    plot_load_analysis(load, time) # Plot that thing.

    # Construct Constrained System
    N = K.shape[0]//3 # Get number of nodes! Note: 3*N = DoF.

    Cc = alphaBeta[0]*Mc + alphaBeta[1]*Kc # Construct the proportional damping␣
↪matrix with pre-determined alpha and beta values.

    f = np.array(np.zeros((3*N, time.shape[0]))) # Initialize load vector array;
↪ Note that the columns contain the force values from 0 to T!
    f[Iz[N1]] = load # Assign load function at point N1 in z-direction.
    fc = f[~Ic] # Reduce load array.

    u0 = np.zeros(3*N) # Initial displacement set to 0.
    u0c = u0[~Ic] # Reduce displacement vector.

    # Time Integration
    uc, vc, ac = Newmark(Mc, Cc ,Kc , fc, time, u0c)
    u = unreduce_constrained(uc, Ic) # Collect the displacement constraints in␣
↪the unreduced displacement array.

    # Plot P1
    plot_P1_timedomain(u, time, N1)

    # Optional animation
```

```
    if display_animation:
        animate_plate_timedomain(u, time, X)
```

[19]:
```
# Check load function
tau = 0.002 # Time constant
t = np.linspace(0, 10*tau, 512) # Use 2^n array length to improve FFT␣
↪performance.
load = gaussianStep(t, tau=0.002) # Load function
plot_load_analysis(load, t) # Plot that thing.
```





6.2.2 Note:

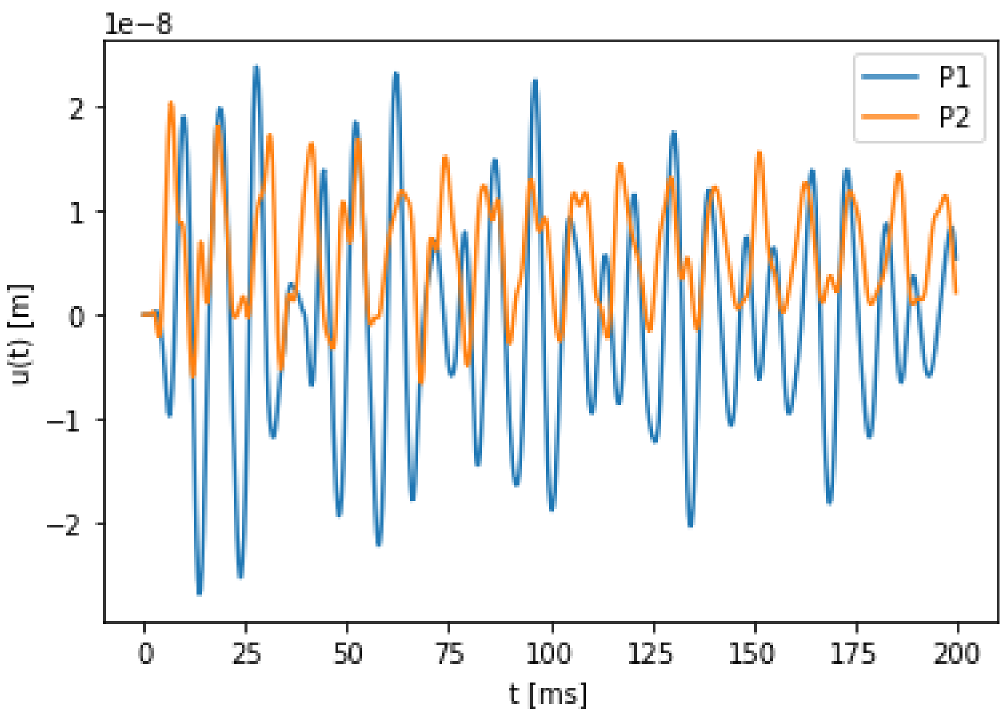
The load signals spectrum reveals that the frequency content spans up to 100 Hz. With this information in mind the integration timestep Δt can be assigned using the rule of thumb

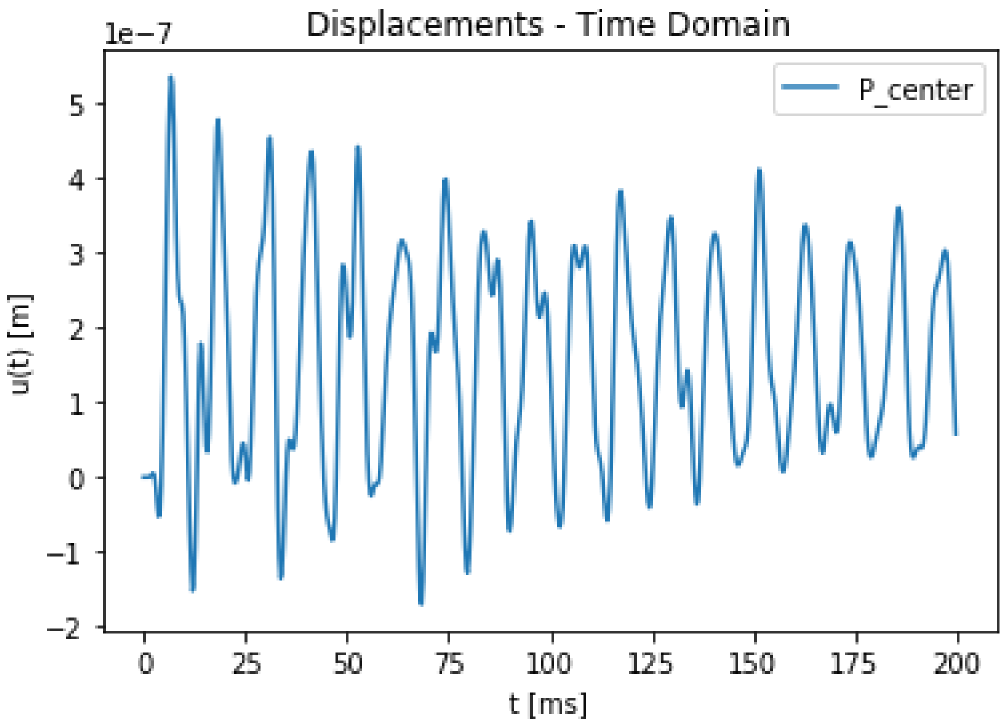
$$\Delta t \leq 1/10 f_{max}.$$

If the timestep is choosen too small, the computational effort becomes infeasible. If it is too big, essential dynamics are not captured.

[20]:
```
# Time
T = 0.2 # Assign right boundary of time interval to 200ms.
f_max = 100 # Max. frequency of load signal
dt = 1/(30*f_max) # Timestep
time = np.arange(0, T, dt) # Create time array for integration

# Construct Constrained System
N = K.shape[0]//3 # Get number of nodes! Note: 3*N = DoF.

Cc = alphaBeta[0]*Mc + alphaBeta[1]*Kc # Construct the proportional damping␣
↪matrix with pre-determined alpha and beta values.

f = np.array(np.zeros((3*N, time.shape[0]))) # Initialize load vector array;␣
↪Note that the columns contain the force values from 0 to T!
```

```
f[Iz[N1]] = gaussianStep(time, tau=0.002) # Assign load function at point P1 in
↪z-direction.
fc = f[~Ic] # Reduce load array.

u0 = np.zeros(3*N) # Initial displacement set to 0.
u0c = u0[~Ic] # Reduce displacement vector.

# Time Integration
uc, vc, ac = Newmark(Mc, Cc ,Kc , fc, time, u0c)
u = unreduce_constrained(uc, Ic) # Collect the displacement constraints in the
↪unreduced displacement array.

# Investigate the response
plot_displacements_timedomain(u, time, N1, N2)
```





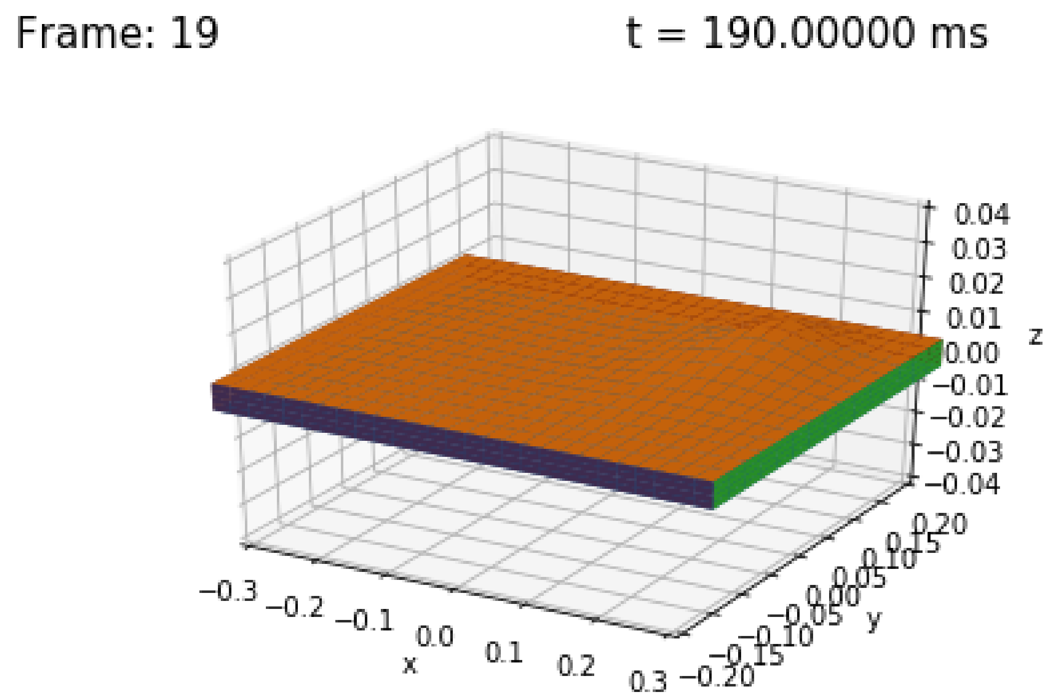
6.3 Animate the transient response

Whe using JSAnimation, be careful not to animate too many time steps, since this might take a long time.

[21]:
```
animate_plate_timedomain(u, time, X)
```

[21]: <matplotlib.animation.FuncAnimation at 0x280017e7348>

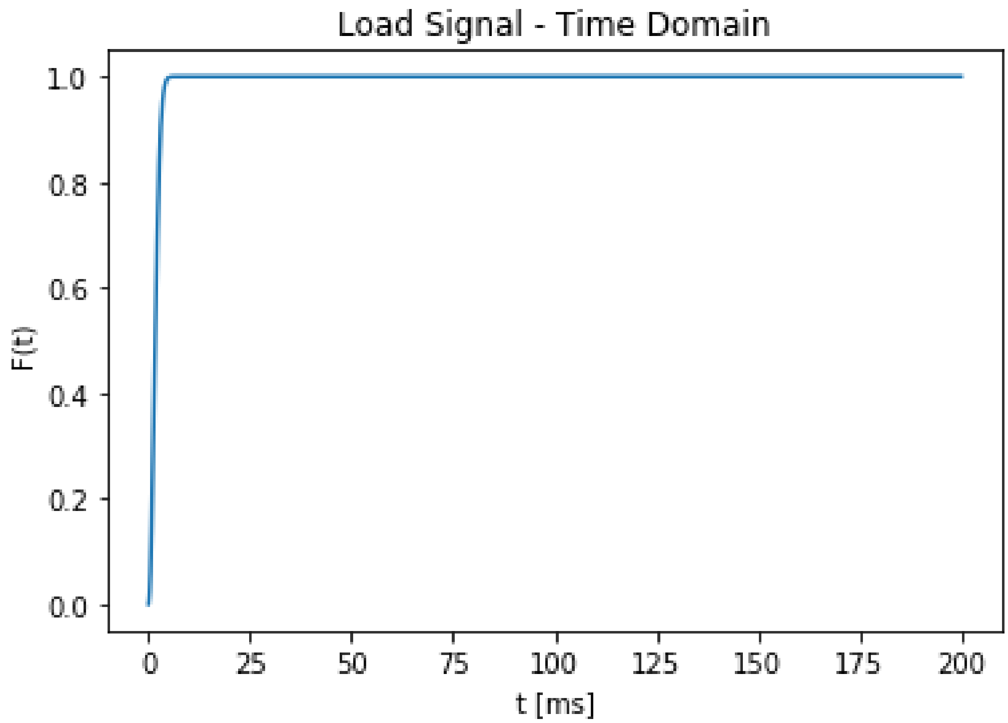
[33]:
```
animate_plate_timedomain(u, time, X, exportFile=True, fileName="transient.mp4")
```



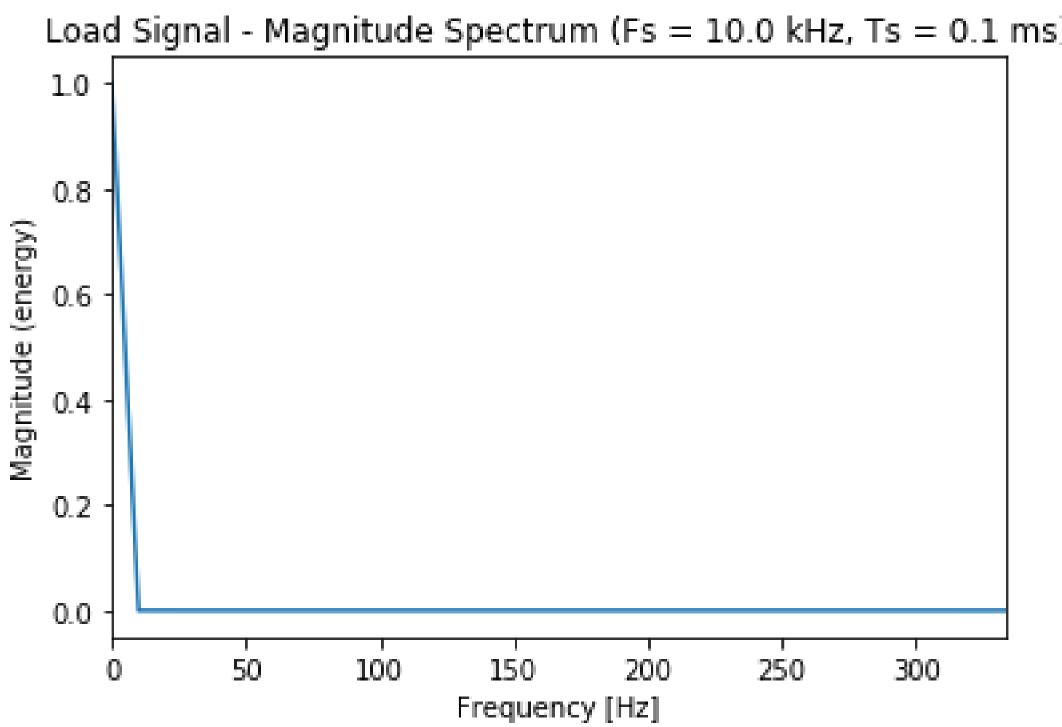
6.4 Compare the response for different forcing functions

Investiagate how the frequency content of the excitation function impacts the output time signal. Plot the z-displacement, e.g. at P1, over time for different excitation signals.

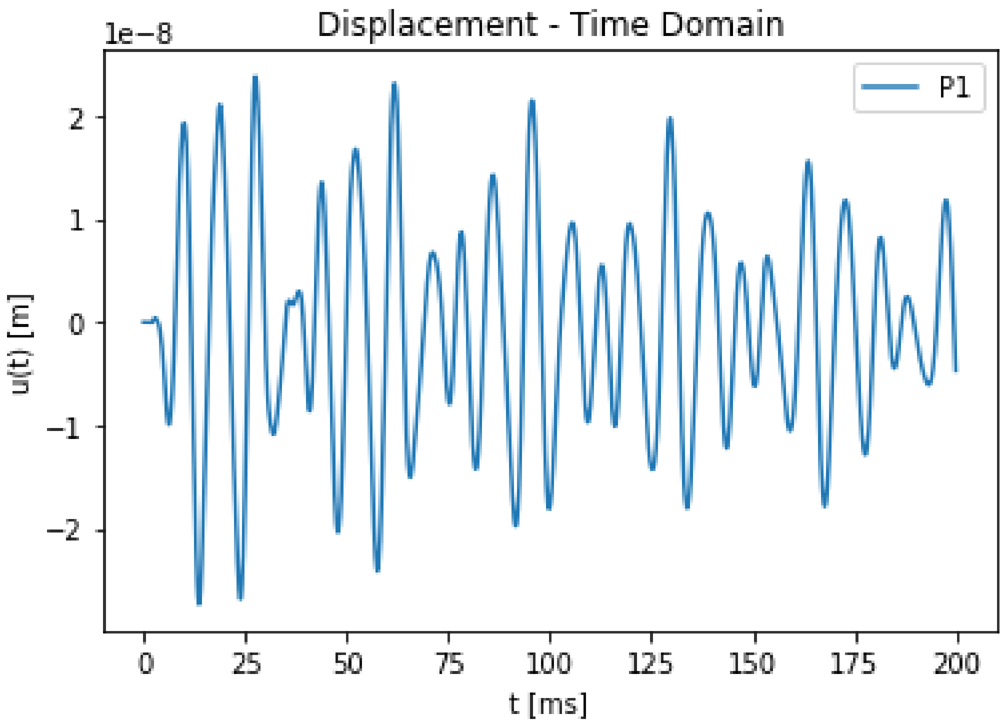
[22]:
```
full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step')
```



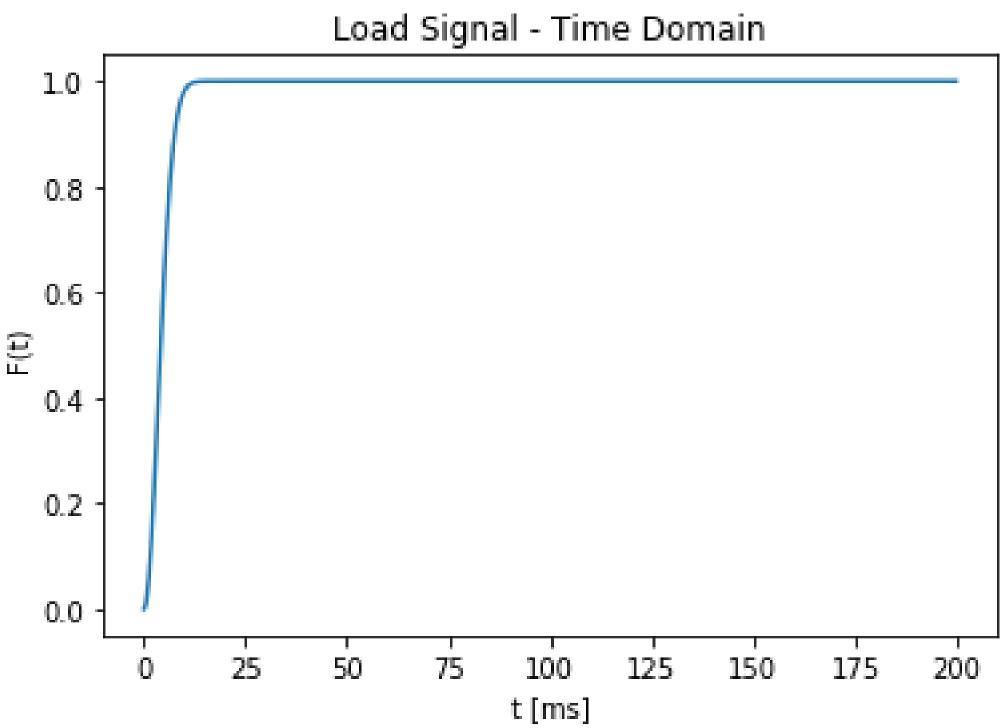
Load Signal - Time Domain
1.0
0.8
0.6
0.4
0.2
0.0
F(t)
0
25
50
75
100
125
150
175
200
t [ms]

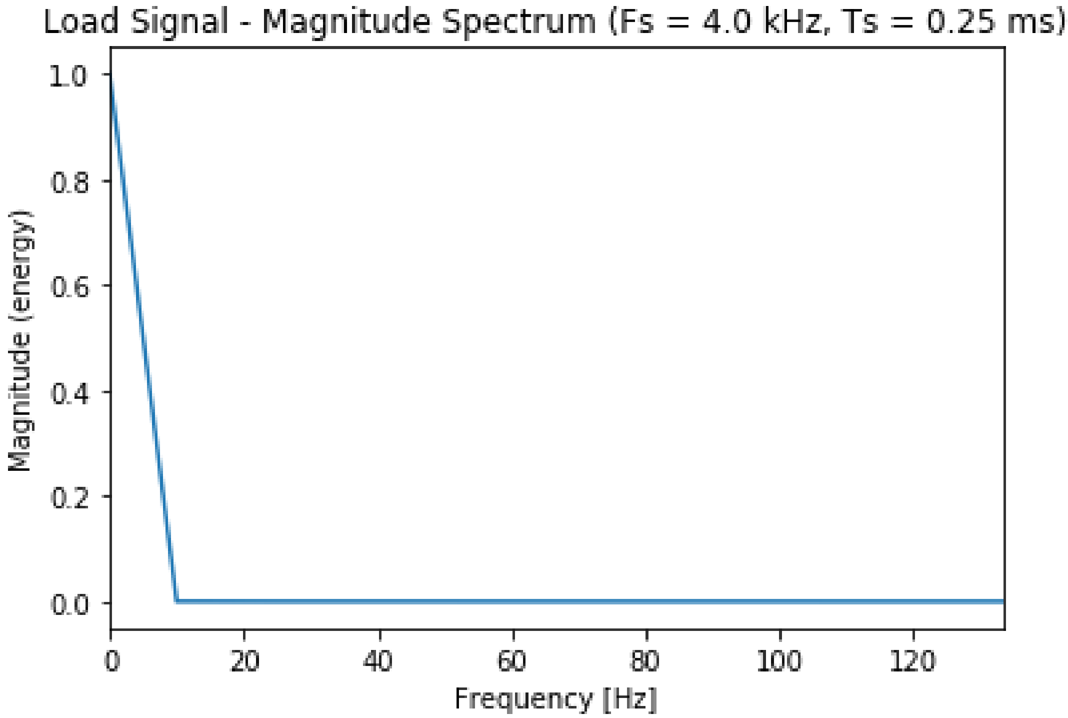


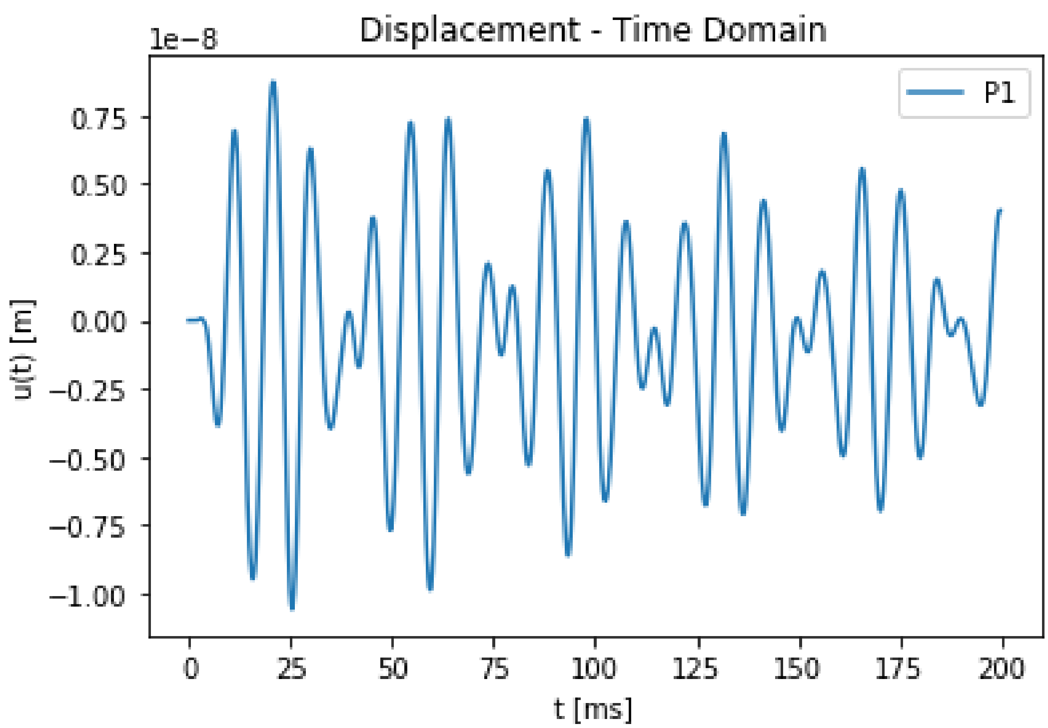
Load Signal - Magnitude Spectrum (Fs = 10.0 kHz, Ts = 0.1 ms)
1.0
0.8
0.6
0.4
0.2
0.0
Magnitude (energy)
0
50
100
150
200
250
300
Frequency [Hz]



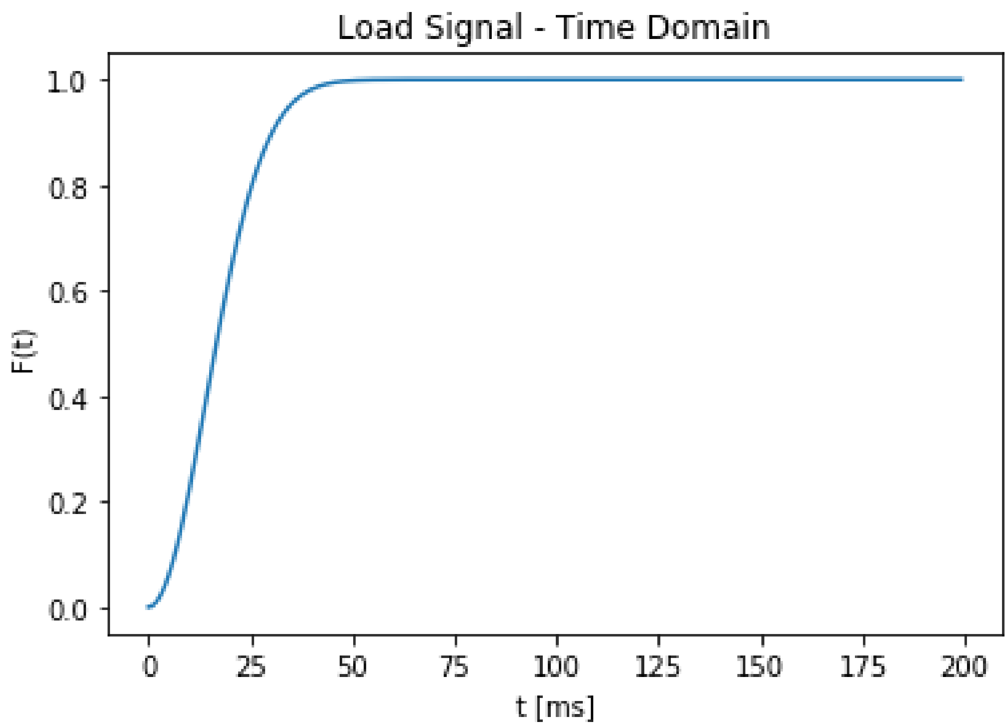
[23]:
```
full_excitation_analysis(tau=0.005, T=0.2, excitation_type='step')
```

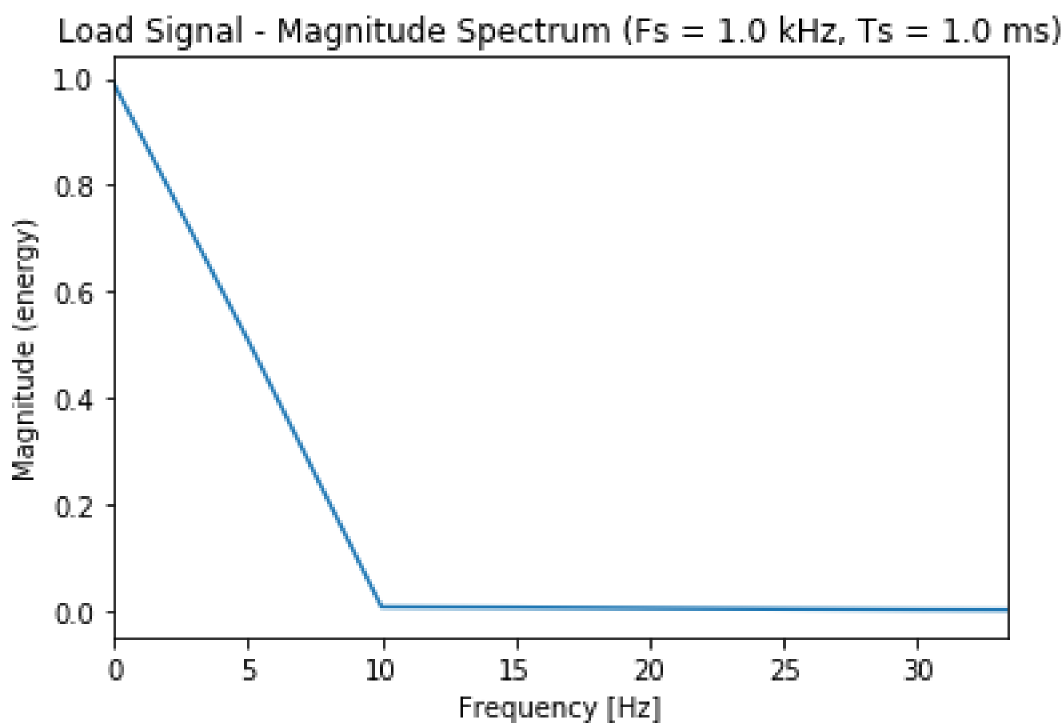


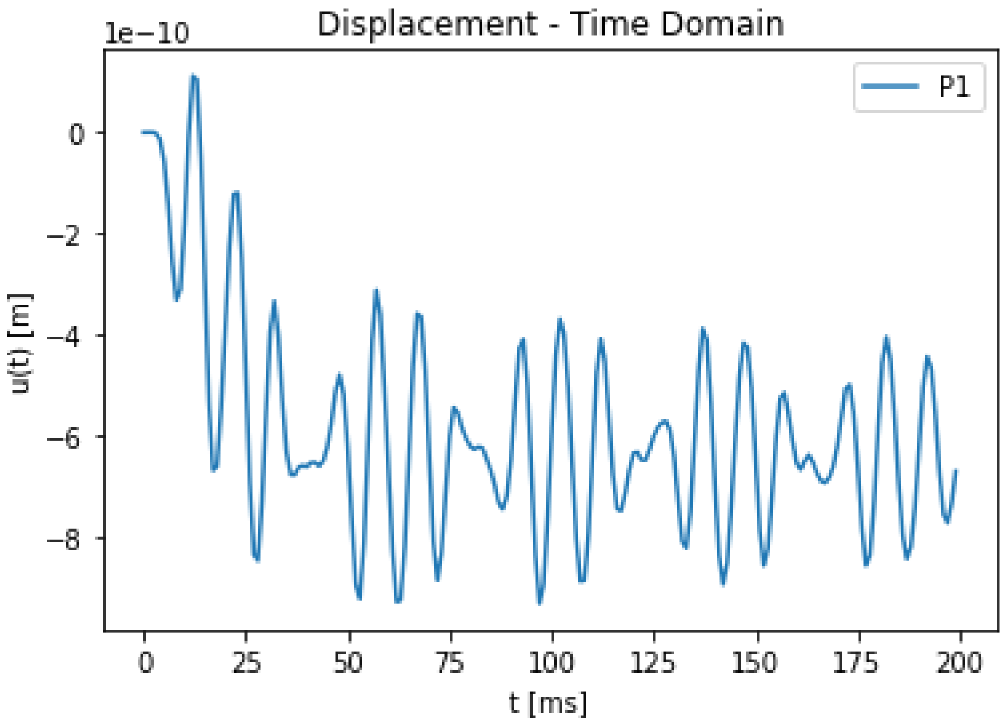

Load Signal - Magnitude Spectrum (Fs = 4.0 kHz, Ts = 0.25 ms)
Magnitude (energy)
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
120
Frequency [Hz]


Displacement - Time Domain
1e−8
P1
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
u(t) [m]
0
25
50
75
100
125
150
175
200
t [ms]

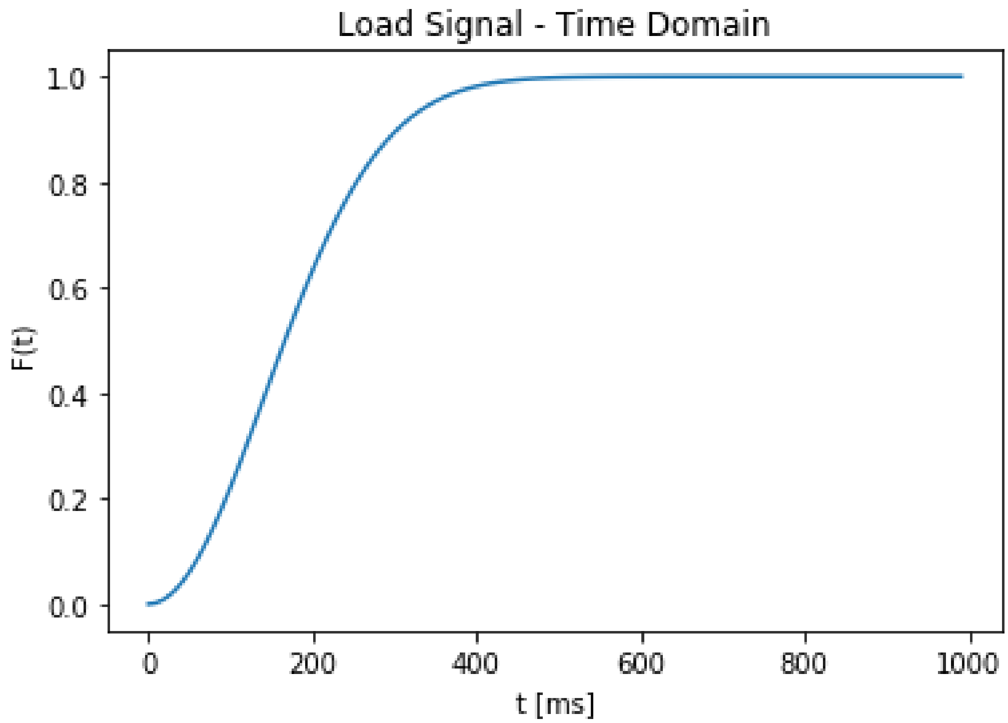
[24]:
```
full_excitation_analysis(tau=0.02, T=0.2, excitation_type='step')
```

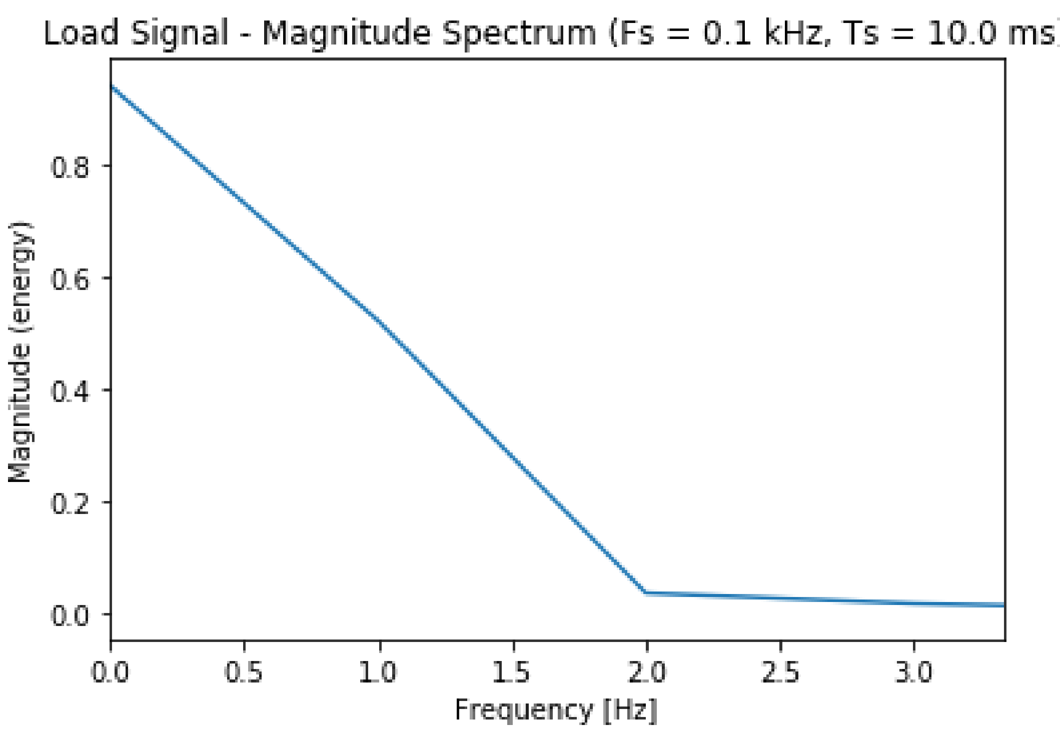


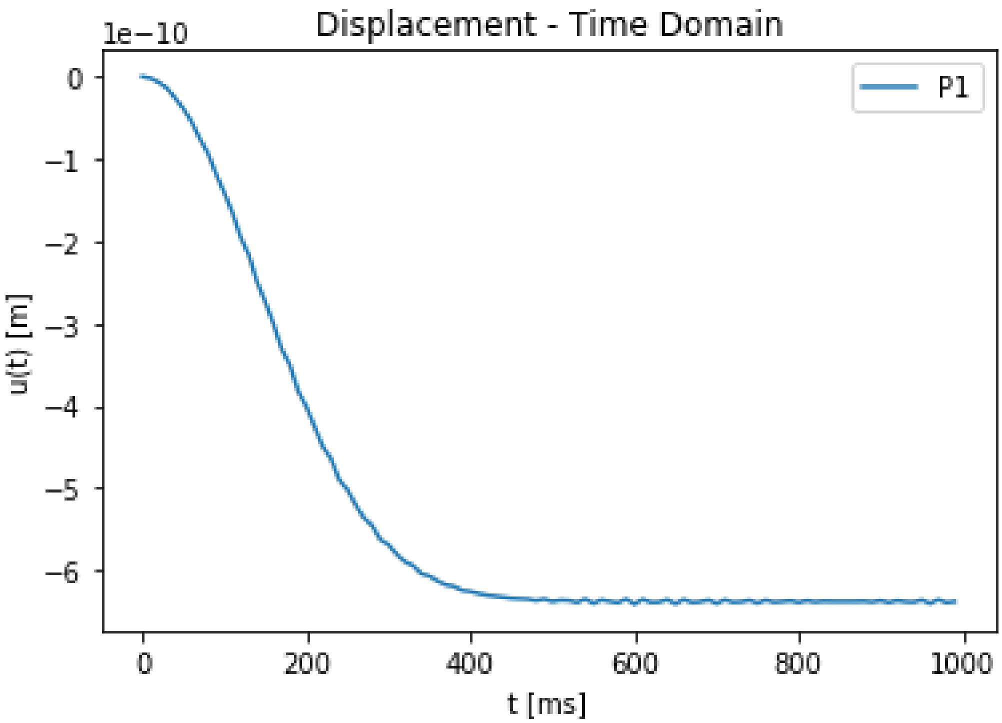





```
[25]: full_excitation_analysis(tau=0.2, T=1, excitation_type='step')
```


Load Signal - Time Domain
1.0
0.8
0.6
0.4
0.2
0.0
F(t)
0
200
400
600
800
1000
t [ms]


Load Signal - Magnitude Spectrum (Fs = 0.1 kHz, Ts = 10.0 ms)
0.8
0.6
0.4
0.2
0.0
Magnitude (energy)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
Frequency [Hz]



7 Frequency domain

7.1 Compute the Steady-State Response

In order to compute the steady state response directly in the frequency domain, we need to 1. Compute the dynamic stiffness matrix for one ω 2. assemble one (or several) forcing vectors 3. solve for the displacements

Use methods for sparse matrices to solve the linear system

from scipy.sparse.linalg import spsolve

[34]:
```
from scipy.sparse.linalg import spsolve
import time
```

[35]:
```
def FrequencyDomain(omega, direc = Iz, node = N1):
    #1. Compute the dynamic stiffness matrix K for one omega
    #Z = Kc.toarray() + complex(0,1) * omega * Cc.toarray() - omega**2 * Mc.
 ↪toarray() #for sparse matrices
    Z = Kc + complex(0,1) * omega * Cc - omega**2 * Mc      #for np.arrays

    #2. Assemble one (or several) forcing vectors
    f_hat = np.zeros(3*N)
```

```
    f_hat[direc[node]] = 1.0     #for sys without constrains and force acting on␣
↪N1 which is the closest node to P1
    fc_hat = f_hat[~Ic]     #for reduced sys, because of constrains

    #3. solve for the displacements
    xc_hat = spsolve(Z,fc_hat)        #for spare matrices
    #x_hat = np.linalg.solve(Z,fc_hat)    #for np.array's

    return(xc_hat) #complex, so ampl and phase is in there; for all DoF which␣
↪are not constrained
```

7.2 Task 2: Transfer function

Compute the steady state response of the system to hamonic forcing in z-direction (unit amplitude) at point P1 in the frequency range from 2Hz to 300Hz (using ~150 frequency points). Assume Rayleigh damping with $\alpha = 2.15$ and $\beta = 0.00003$.

Plot the response (amplitude and phase) for the z-diplacement at points P1 and P2, as well as for the center of the plate.

[36]:
```
start_time = time.time()
#assemble Damping-Matrix for the reduced sys and given aplha and beta for␣
↪Rayleigh damping
alpha = 2.15
beta = 0.00003
Cc = alpha * Mc + beta * Kc

#find node number in the center
PC = [0.,0.,0.]
NC = np.argmin(np.sum((X-PC)**2,axis=1))

#frequency range from 2Hz to 300Hz and ~150 frequency points
Nr_steps = 150.
steps = (300.-2.)/Nr_steps

P1_resp_z = np.zeros([int(Nr_steps-1.),2]) #+ complex(0,0)
P2_resp_z = np.zeros([int(Nr_steps-1.),2]) #+ complex(0,0)
PC_resp_z = np.zeros([int(Nr_steps-1.),2]) #+ complex(0,0)

counter = 0

for i in range(2, 300, round(steps)):
    resp = FrequencyDomain(2*np.pi*i)

    # insert missing nodes with zero
    resp_all = np.zeros(N*3) + complex(0,0)
    resp_all[~Ic] = resp
```

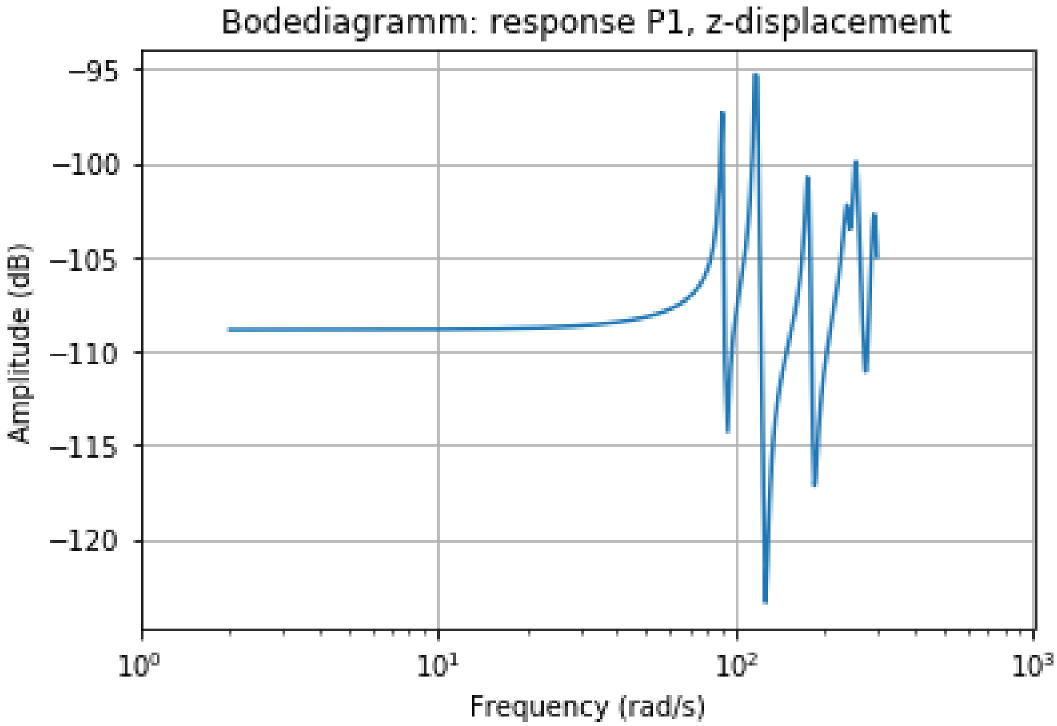
```
    #Amplitude in dB
    P1_resp_z[counter,0] = 20*np.log10(np.abs(resp_all[Iz[N1]]))
    P2_resp_z[counter,0] = 20*np.log10(np.abs(resp_all[Iz[N2]]))
    PC_resp_z[counter,0] = 20*np.log10(np.abs(resp_all[Iz[NC]]))
    #Phase in degree
    P1_resp_z[counter,1] = np.angle(resp_all[Iz[N1]])*180/np.pi
    P2_resp_z[counter,1] = np.angle(resp_all[Iz[N2]])*180/np.pi
    PC_resp_z[counter,1] = np.angle(resp_all[Iz[NC]])*180/np.pi

    counter += 1

print("--- %s seconds ---" % (time.time() - start_time))
```

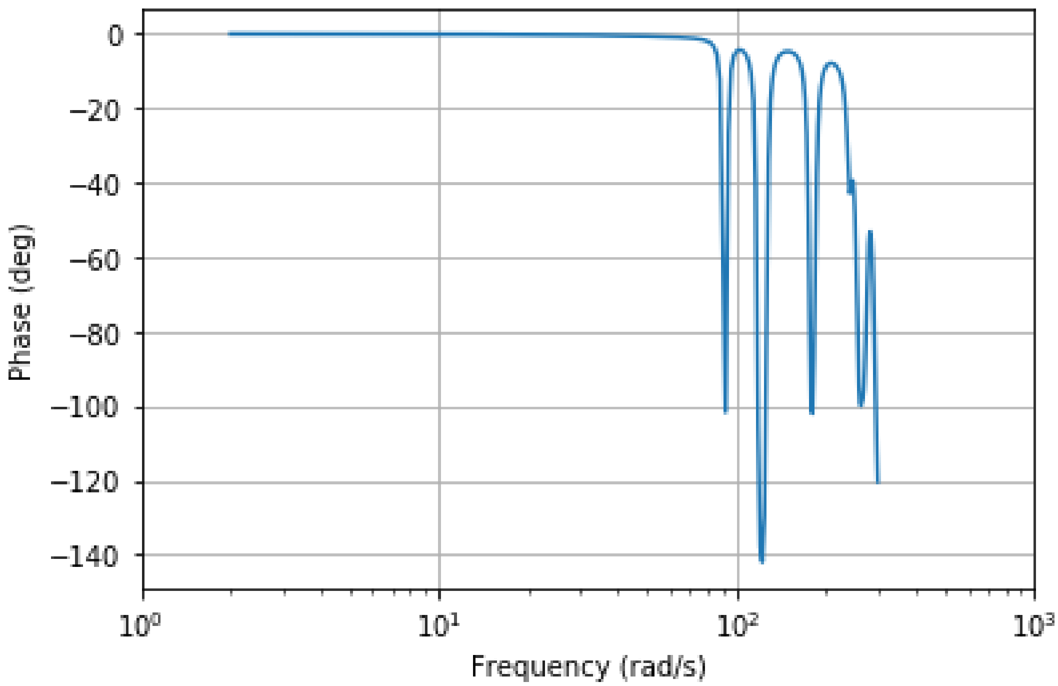
```
--- 131.49671959877014 seconds ---
```

[37]:
```
#array with associate frequency values, for Bode-Diagramms
frequency = range(2, 300 , round(steps))
```

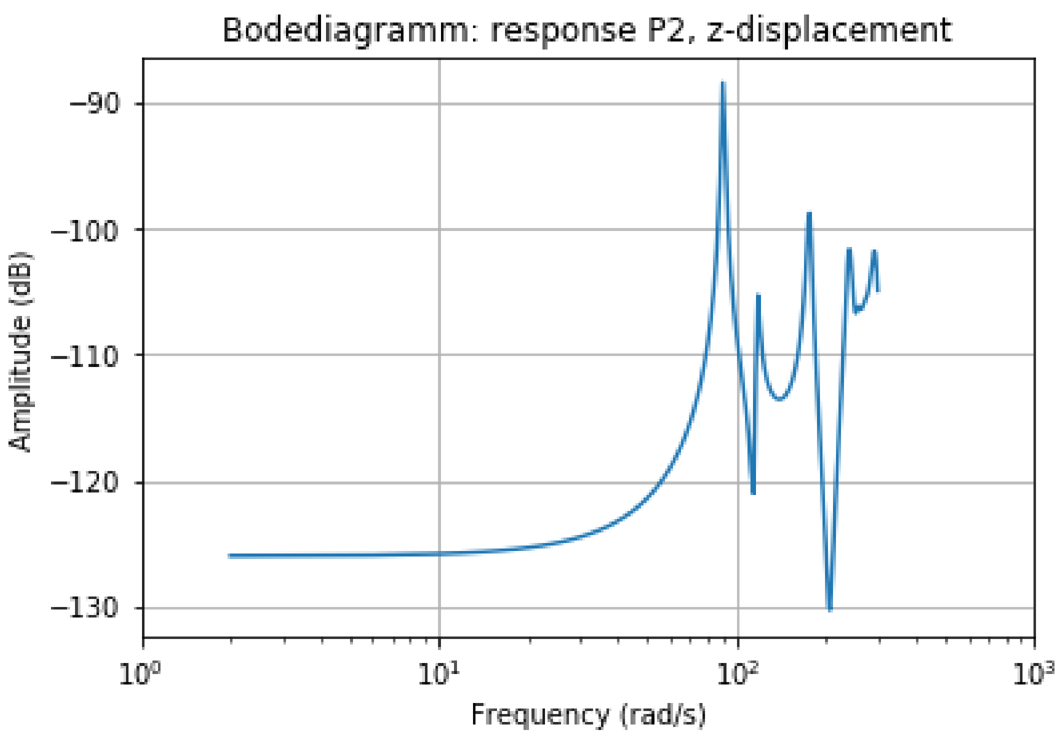
[38]:
```
#plot response in z for P1
plt.plot(frequency, P1_resp_z[:,0])
plt.title('Bodediagramm: response P1, z-displacement')
plt.ylabel('Amplitude (dB)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()

plt.plot(frequency, P1_resp_z[:,1])
plt.ylabel('Phase (deg)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()

#plot response in z for P2
plt.plot(frequency, P2_resp_z[:,0])
plt.title('Bodediagramm: response P2, z-displacement')
plt.ylabel('Amplitude (dB)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()
```

```
plt.plot(frequency, P2_resp_z[:,1])
plt.ylabel('Phase (deg)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()

#plot response in z for PC
plt.plot(frequency, PC_resp_z[:,0])
plt.title('Bodediagramm: response PC, z-displacement')
plt.ylabel('Amplitude (dB)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()

plt.plot(frequency, PC_resp_z[:,1])
plt.ylabel('Phase (deg)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()
```

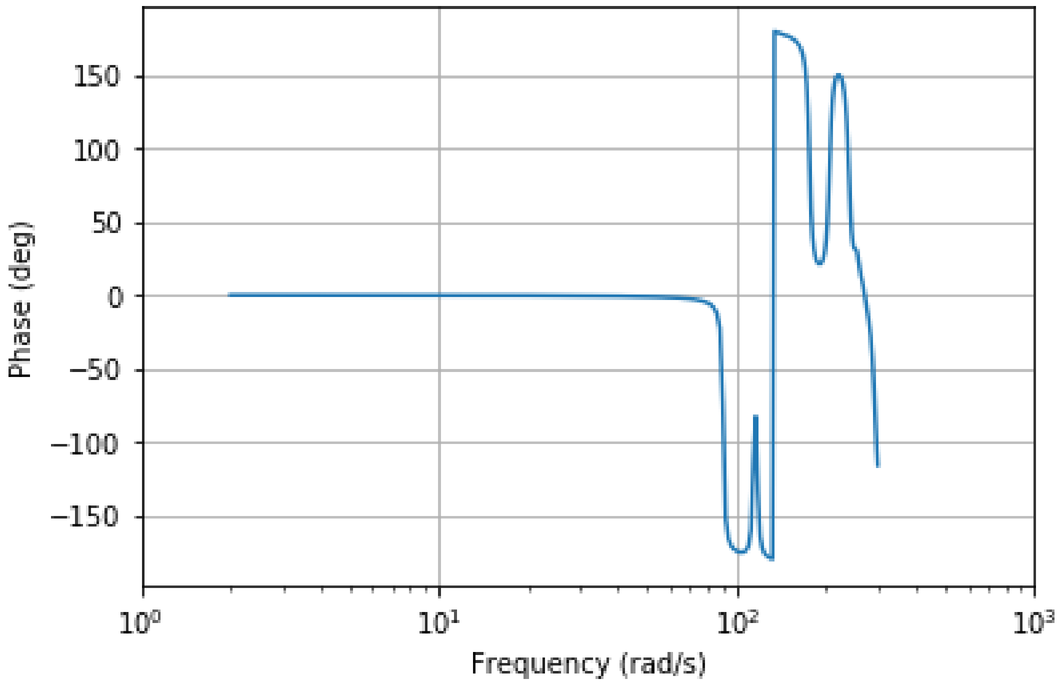




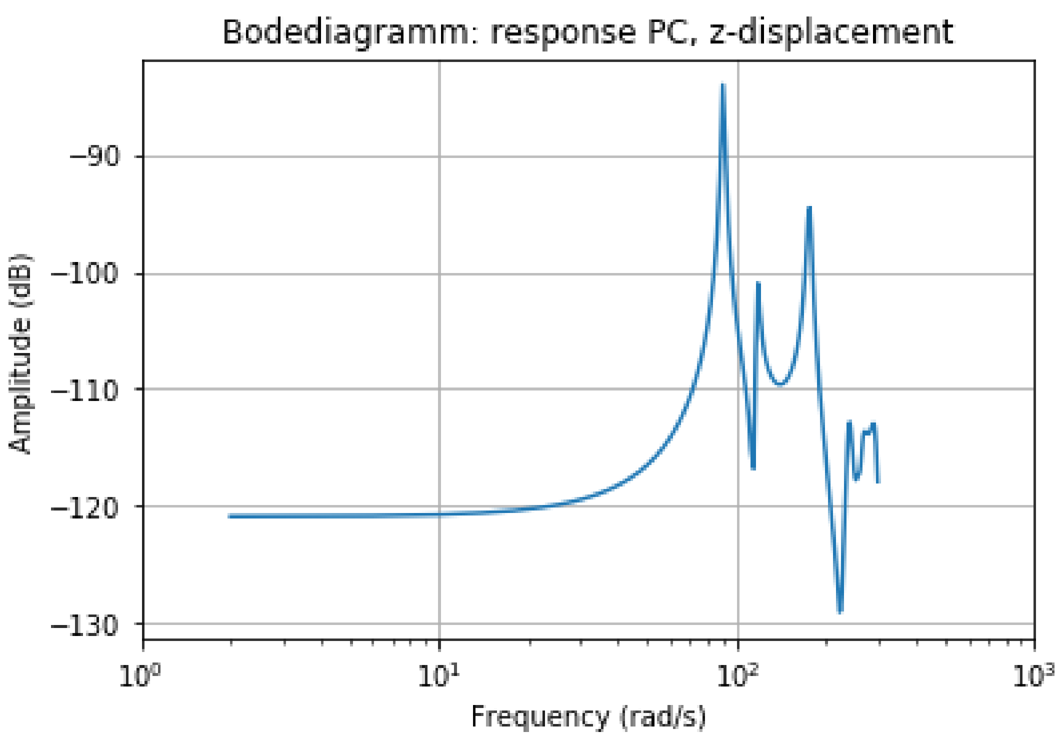
Phase (deg)
0
−20
−40
−60
−80
−100
−120
−140
10^0
10^1
10^2
10^3
Frequency (rad/s)



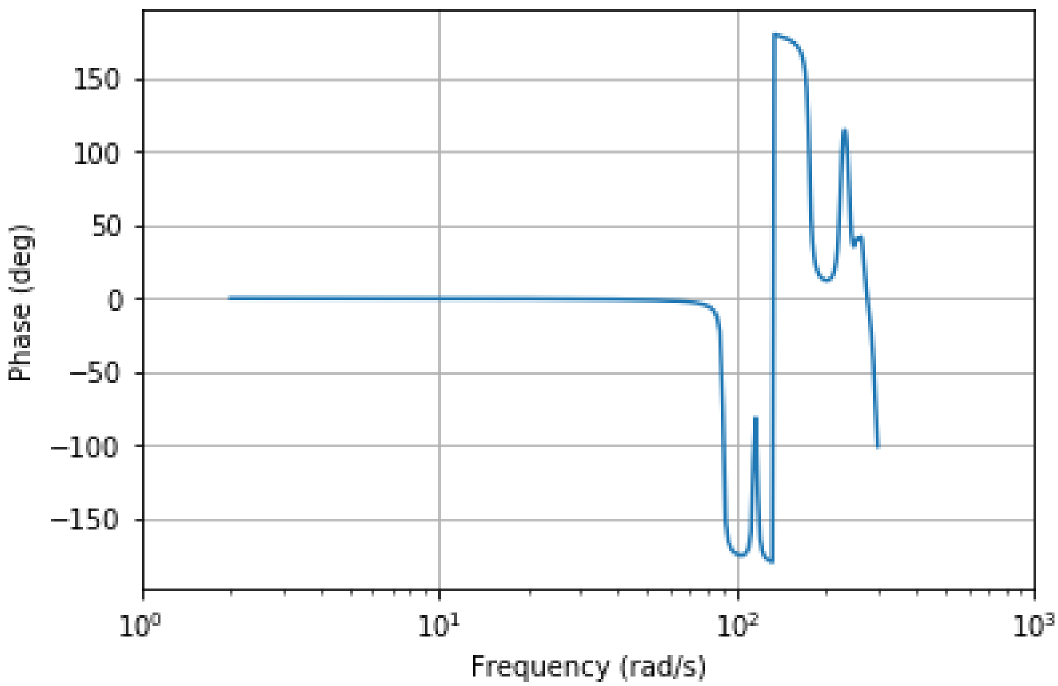
Bodediagramm: response P2, z-displacement
Amplitude (dB)
−90
−100
−110
−120
−130
10^0
10^1
10^2
10^3
Frequency (rad/s)



150
100
50
0
−50
−100
−150
Phase (deg)
10^0
10^1
10^2
10^3
Frequency (rad/s)



Bodediagramm: response PC, z-displacement
−90
−100
−110
−120
−130
Amplitude (dB)
10^0
10^1
10^2
10^3
Frequency (rad/s)



7.3 Animate the harmonic response

You can use the same function as for animating mode shapes. Look the response at characteristic frequecy points, e.g. at the peaks or minima of the transfer function.

[39]:
```
def animateFrequencyResponse(Frequency,whatTOdo):
    plt.rcParams["animation.html"] = "jshtml"

    resp = FrequencyDomain(2*np.pi*Frequency)

    u = np.zeros(3*N) # initialize displacement vector
    u[~Ic] = np.abs(resp)

    # format U like X
    U = np.array([u[Ix],u[Iy],u[Iz]]).T

    #------------------------------------------------------------
    # set up figure and animation
    fig, ax = plt.subplots(subplot_kw={'projection':'3d'})

    # Plot a basic wireframe.
    index = 1

    x_bot = np.reshape(X[Nb,0],(len(y),len(x)))
```

```
y_bot = np.reshape(X[Nb,1],(len(y),len(x)))
z_bot = np.reshape(X[Nb,2],(len(y),len(x)))

x_top = np.reshape(X[Nt,0],(len(y),len(x)))
y_top = np.reshape(X[Nt,1],(len(y),len(x)))
z_top = np.reshape(X[Nt,2],(len(y),len(x)))

x_o = np.reshape(X[No,0],(len(y),len(z)))
y_o = np.reshape(X[No,1],(len(y),len(z)))
z_o = np.reshape(X[No,2],(len(y),len(z)))

x_n = np.reshape(X[Nn,0],(len(x),len(z)))
y_n = np.reshape(X[Nn,1],(len(x),len(z)))
z_n = np.reshape(X[Nn,2],(len(x),len(z)))

x_s = np.reshape(X[Ns,0],(len(x),len(z)))
y_s = np.reshape(X[Ns,1],(len(x),len(z)))
z_s = np.reshape(X[Ns,2],(len(x),len(z)))

x_w = np.reshape(X[Nw,0],(len(y),len(z)))
y_w = np.reshape(X[Nw,1],(len(y),len(z)))
z_w = np.reshape(X[Nw,2],(len(y),len(z)))

sf1 = ax.plot_surface(x_bot, y_bot, z_bot, rstride=index, cstride=index)
sf2 = ax.plot_surface(x_top, y_top, z_top, rstride=index, cstride=index)
sf3 = ax.plot_surface(x_o, y_o, z_o, rstride=index, cstride=index)
sf4 = ax.plot_surface(x_n, y_n, z_n, rstride=index, cstride=index)
sf5 = ax.plot_surface(x_s, y_s, z_s, rstride=index, cstride=index)
sf6 = ax.plot_surface(x_w, y_w, z_w, rstride=index, cstride=index)

ax.set_xlim(-0.3,0.3)
ax.set_ylim(-0.2,0.2)
ax.set_zlim(-0.04,0.04)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

def animate(i):
    # scale factor for plotting
    s = 0.5/np.max(np.sqrt(np.sum(U**2,axis=0)))*np.sin(1/25*2*np.pi*i)
    Xu = X + s*U # defomed configuration (displacement scaled by s)

    x_bot = np.reshape(Xu[Nb,0],(len(y),len(x)))
    y_bot = np.reshape(Xu[Nb,1],(len(y),len(x)))
    z_bot = np.reshape(Xu[Nb,2],(len(y),len(x)))

    x_top = np.reshape(Xu[Nt,0],(len(y),len(x)))
```

```
        y_top = np.reshape(Xu[Nt,1],(len(y),len(x)))
        z_top = np.reshape(Xu[Nt,2],(len(y),len(x)))

        x_o = np.reshape(Xu[No,0],(len(y),len(z)))
        y_o = np.reshape(Xu[No,1],(len(y),len(z)))
        z_o = np.reshape(Xu[No,2],(len(y),len(z)))

        x_n = np.reshape(Xu[Nn,0],(len(x),len(z)))
        y_n = np.reshape(Xu[Nn,1],(len(x),len(z)))
        z_n = np.reshape(Xu[Nn,2],(len(x),len(z)))

        x_s = np.reshape(Xu[Ns,0],(len(x),len(z)))
        y_s = np.reshape(Xu[Ns,1],(len(x),len(z)))
        z_s = np.reshape(Xu[Ns,2],(len(x),len(z)))

        x_w = np.reshape(Xu[Nw,0],(len(y),len(z)))
        y_w = np.reshape(Xu[Nw,1],(len(y),len(z)))
        z_w = np.reshape(Xu[Nw,2],(len(y),len(z)))

        ax.clear()
        sf1 = ax.plot_surface(x_bot,y_bot,z_bot,rstride=index, cstride=index)
        sf2 = ax.plot_surface(x_top,y_top,z_top,rstride=index, cstride=index)
        sf3 = ax.plot_surface(x_o,y_o,z_o,rstride=index, cstride=index)
        sf4 = ax.plot_surface(x_n,y_n,z_n,rstride=index, cstride=index)
        sf5 = ax.plot_surface(x_s,y_s,z_s,rstride=index, cstride=index)
        sf6 = ax.plot_surface(x_w,y_w,z_w,rstride=index, cstride=index)

        ax.set_xlim(-0.3,0.3)
        ax.set_ylim(-0.2,0.2)
        ax.set_zlim(-0.04,0.04)
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')

        return sf1,sf2,sf3,sf5,sf6,sf4

    ani = animation.FuncAnimation(fig, animate, frames=50, interval=100)

    # # save the animation as an mp4.  This requires ffmpeg or mencoder to be
    # # installed.  The extra_args ensure that the x264 codec is used, so that
    # # the video can be embedded in html5.  You may need to adjust this for
    # # your system: for more information, see
    # # http://matplotlib.sourceforge.net/api/animation_api.html

    if whatTOdo == 'Save' :
        ani.save('modalanalyse_mode_' + str(i) + '.mp4', fps=30,␣
↪extra_args=['-vcodec', 'libx264'])
```

```
    elif whatTOdo == 'justShow':
        plt.close()
        return ani
```

[40]:
```
alpha = 2.15
beta = 0.00003
Cc = alpha * Mc + beta * Kc

animateFrequencyResponse(80,'justShow')
```

[40]: `<matplotlib.animation.FuncAnimation at 0x2800c4c5448>`

7.4 Compare damping

Compute the steady state response of the system to hamonic forcing (as above) for the un-damped system, as well as for the two Rayleigh damping models mentioned above. Compare the transfer functions for the z-displacement of P1.

[42]:
```
#RAYLEIGH NUMBER 1: alpha=4.186645341745284 and beta=4.6173670560012426e-06
start_time = time.time()

#assemble Damping-Matrix for un-damped system
alpha = 4.186645341745284
beta = 4.6173670560012426e-06
Cc = alpha * Mc + beta * Kc

#frequency range from 2Hz to 300Hz and ~150 frequency points
Nr_steps = 150.
steps = (300.-2.)/Nr_steps

P1_resp_z_rayleigh = np.zeros([int(Nr_steps-1.),2]) #+ complex(0,0)

counter = 0

for i in range(2, 300, round(steps)):
    resp = FrequencyDomain(2*np.pi*i)

    # insert missing nodes with zero
    resp_all = np.zeros(N*3) + complex(0,0)
    resp_all[~Ic] = resp

    #Amplitude in dB
    P1_resp_z_rayleigh[counter,0] = 20*np.log10(np.abs(resp_all[Iz[N1]]))

    #Phase in deg
    P1_resp_z_rayleigh[counter,1] = np.angle(resp_all[Iz[N1]])*180/np.pi
```

```
    counter += 1

print("--- %s seconds ---" % (time.time() - start_time))
```

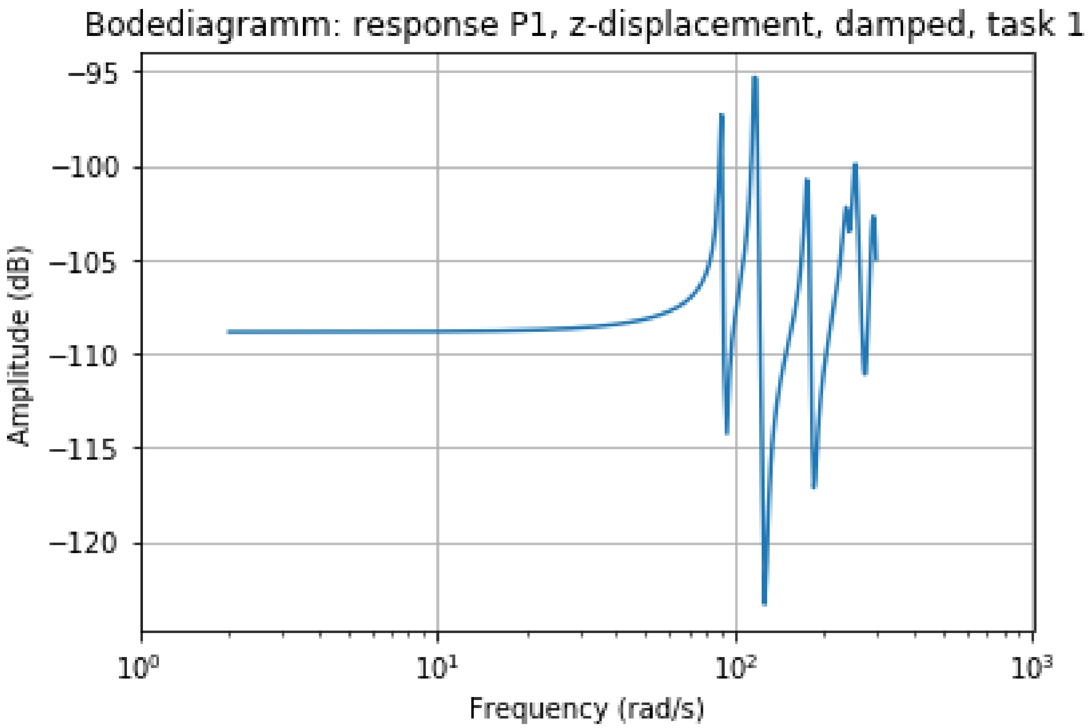
```
--- 132.96803283691406 seconds ---
```

[43]:
```
#UNDAMPED
start_time = time.time()

#assemble Damping-Matrix for un-damped system
Cc = csc_matrix(C[np.ix_(~Ic,~Ic)]) # Replace this for other Rayleigh damping␣
 ↪models

#frequency range from 2Hz to 300Hz and ~150 frequency points
Nr_steps = 150.
steps = (300.-2.)/Nr_steps

P1_resp_z_undamped = np.zeros([int(Nr_steps-1.),2]) #+ complex(0,0)

counter = 0

for i in range(2, 300, round(steps)):
    resp = FrequencyDomain(2*np.pi*i)

    # insert missing nodes with zero
    resp_all = np.zeros(N*3) + complex(0,0)
    resp_all[~Ic] = resp

    #Amplitude in dB
    P1_resp_z_undamped[counter,0] = 20*np.log10(np.abs(resp_all[Iz[N1]]))

    #Phase in deg
    P1_resp_z_undamped[counter,1] = np.angle(resp_all[Iz[N1]])*180/np.pi

    counter += 1

print("--- %s seconds ---" % (time.time() - start_time))
```

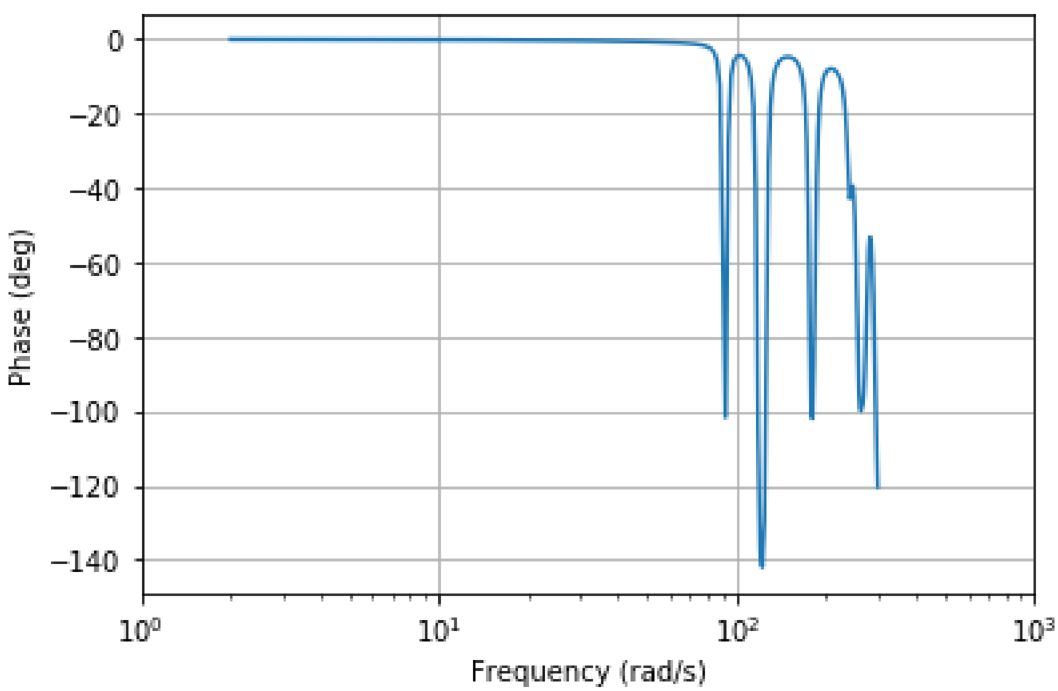
```
--- 132.87795162200928 seconds ---
```

[44]:
```
#plot response in z for P1, damped sys. task2
plt.plot(frequency, P1_resp_z[:,0])
plt.title('Bodediagramm: response P1, z-displacement, damped, task 1')
plt.ylabel('Amplitude (dB)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
```

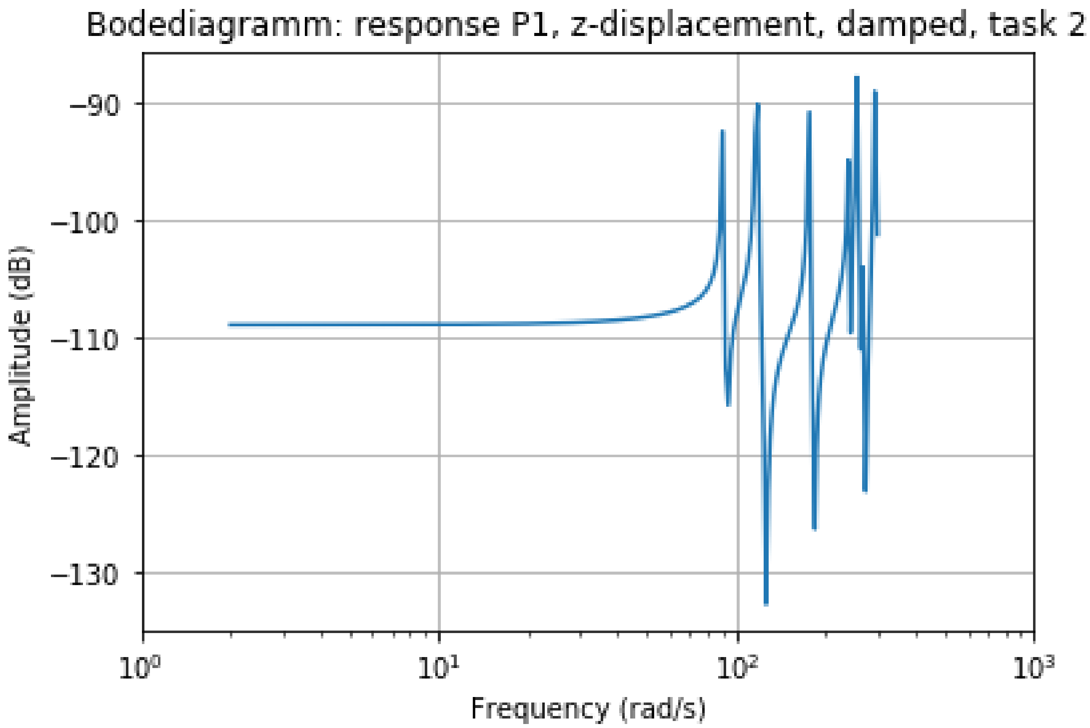
```python
plt.grid(True)
plt.show()

plt.plot(frequency, P1_resp_z[:,1])
plt.ylabel('Phase (deg)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()

#plot response in z for P1, damped sys. task2
plt.plot(frequency, P1_resp_z_rayleigh[:,0])
plt.title('Bodediagramm: response P1, z-displacement, damped, task 2')
plt.ylabel('Amplitude (dB)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()

plt.plot(frequency, P1_resp_z_rayleigh[:,1])
plt.ylabel('Phase (deg)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()

#plot response in z for P1, undamped sys.
plt.plot(frequency, P1_resp_z_undamped[:,0])
plt.title('Bodediagramm: response P1, z-displacement, undamped')
plt.ylabel('Amplitude (dB)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()

plt.plot(frequency, P1_resp_z_undamped[:,1])
plt.ylabel('Phase (deg)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.show()
```



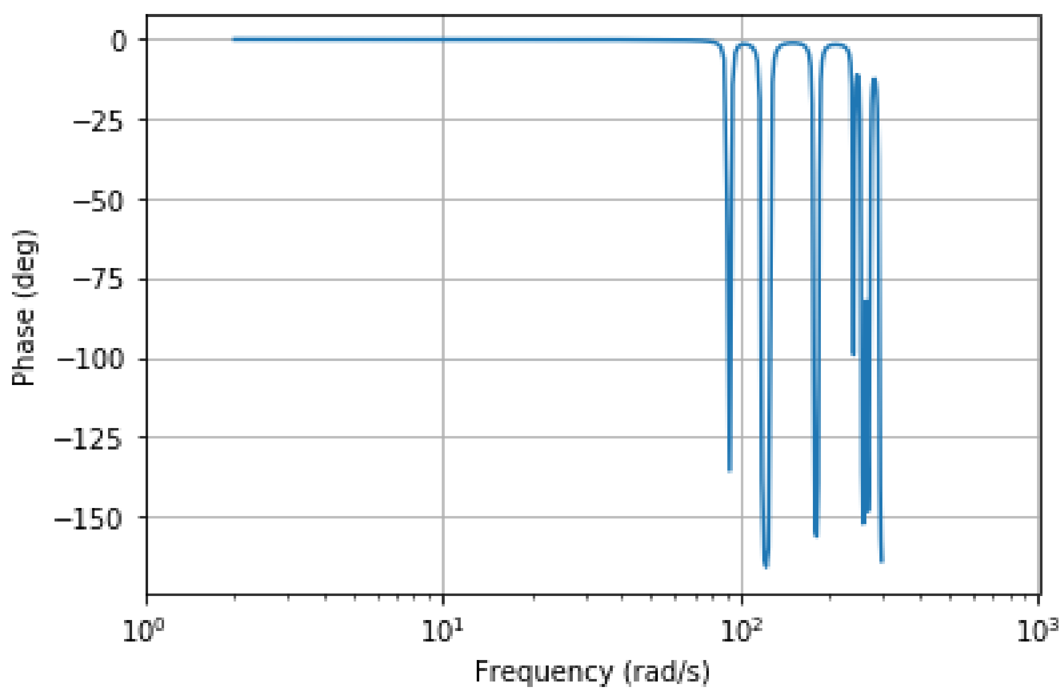
Bodediagramm: response P1, z-displacement, damped, task 1
−95
−100
−105
−110
−115
−120
Amplitude (dB)
10^0
10^1
10^2
10^3
Frequency (rad/s)



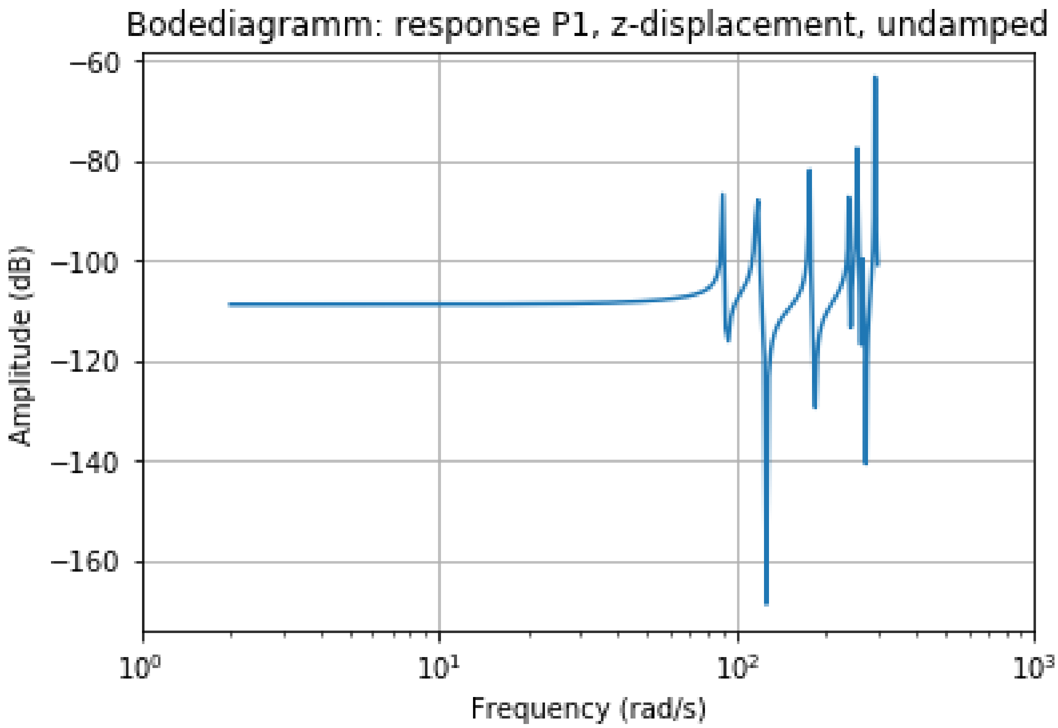
0
−20
−40
−60
−80
−100
−120
−140
Phase (deg)
10^0
10^1
10^2
10^3
Frequency (rad/s)

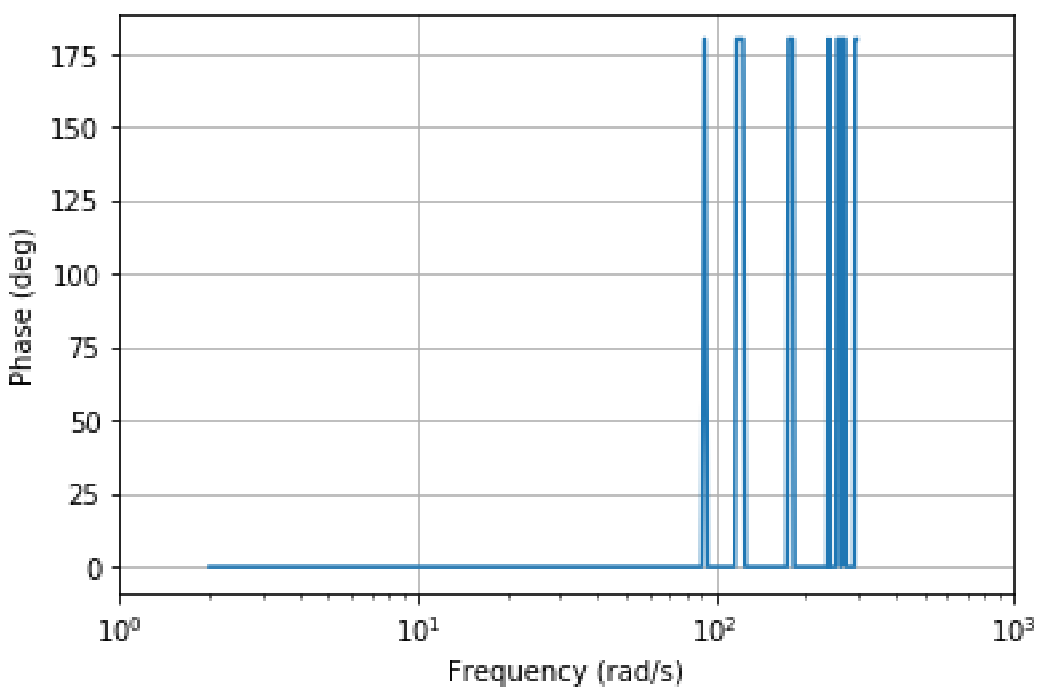


Bodediagramm: response P1, z-displacement, damped, task 2
Amplitude (dB)
−90
−100
−110
−120
−130
10^0
10^1
10^2
10^3
Frequency (rad/s)



Phase (deg)
0
−25
−50
−75
−100
−125
−150
10^0
10^1
10^2
10^3
Frequency (rad/s)


Bodediagramm: response P1, z-displacement, undamped
Amplitude (dB)
−60
−80
−100
−120
−140
−160
10^0
10^1
10^2
10^3
Frequency (rad/s)


Phase (deg)
175
150
125
100
75
50
25
0
10^0
10^1
10^2
10^3
Frequency (rad/s)

7.5 Estimate transfer function from modal data

Compute the un-damped transfer function (Receptance matrix) using the modal parameters (mode shape matrix and natural frequencies). Compare this estimate to the transfer functions computed above. What about the modal estimate using only 2 modes?

The recepance matrix is a large dense matrix 3N x 3N. Do not try to store it for many frequency values. Only compute and store the elements you need.

[45]:
```
# Use W and V from previous calc
k = 50
W,V = eigsh(Kc,k,Mc,sigma=0,which='LM',maxiter = 1000) # takes way to long to␣
↪find samllest values

def receptanceMatrix(V,W,omega) :
    container = np.array(1/(W - omega**2))
    diagMiddle = np.diag(container)
    H = V @ diagMiddle @ V.transpose()
    return(H)
```

[46]:
```
print("Max frequency %f Hz" % (np.sqrt(abs(W[-1]))/2/np.pi))
```

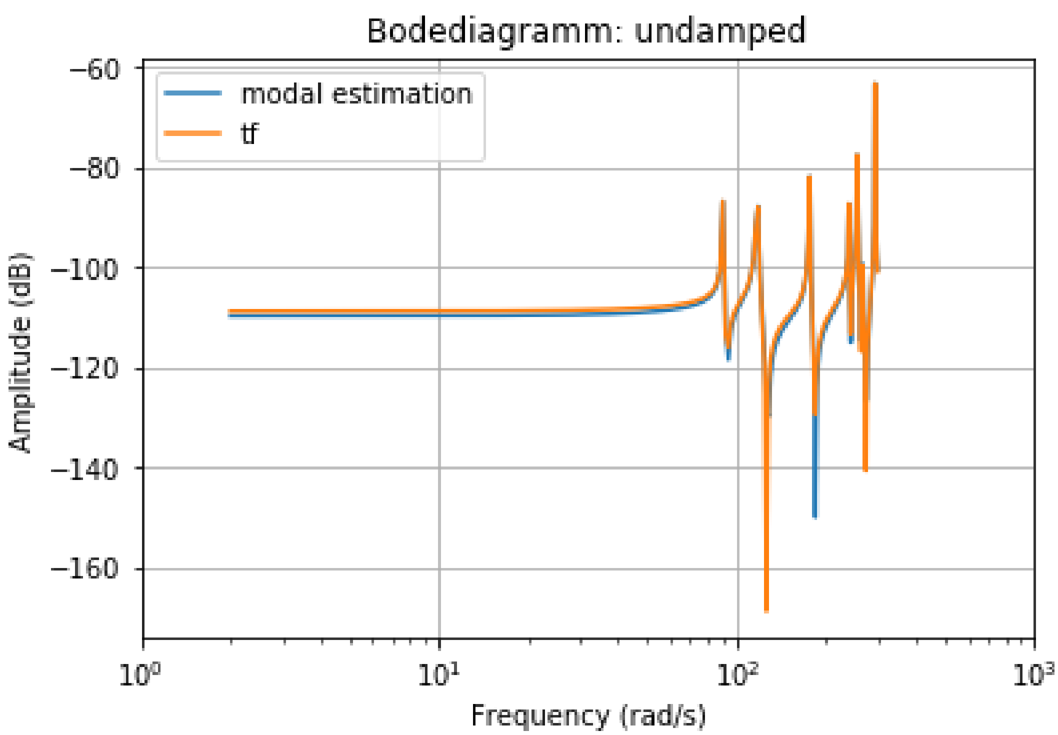
```
Max frequency 1466.892284 Hz
```

[47]:
```
start_time = time.time()

f_hat = np.zeros(3*N)
f_hat[Iz[N1]] = 1.0    #for sys without constrains and force acting on N1 which␣
↪is the closest node to P1
fc_hat = f_hat[~Ic]    #for reduced sys, because of constrains

# Steps for calc
Nr_steps = 150.
steps = (300.-2.)/Nr_steps

P1_resp_z_undamped_modal = np.array([])

for i in range(2, 300, round(steps)):
    omega = 2*np.pi*i
    H = receptanceMatrix(V,W,omega)
    x_hat = H @ fc_hat

    # insert missing nodes with zero
    x_hat_all = np.zeros(N*3) + complex(0,0)
    x_hat_all[~Ic] = x_hat

    P1_resp_z_undamped_modal = np.concatenate( ( P1_resp_z_undamped_modal,␣
↪20*np.log10(np.array([np.abs(x_hat_all[Iz[N1]])]))) )
```

```
print("--- %s seconds ---" % (time.time() - start_time))
```

--- 16.834583044052124 seconds ---

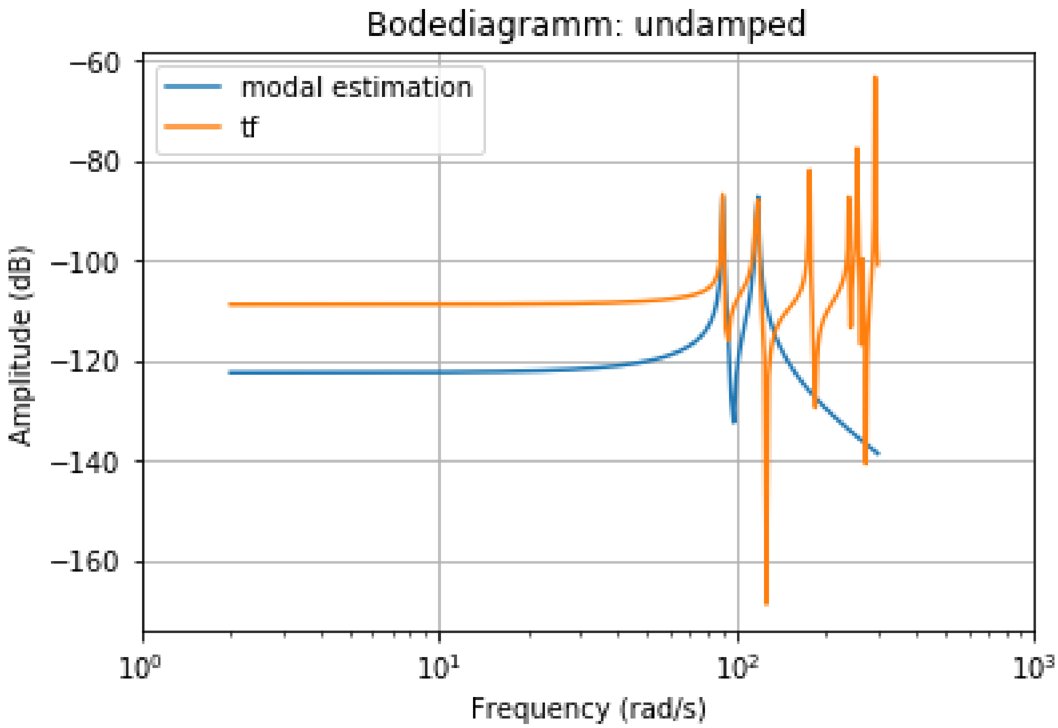
[48]:
```
#plot response in z for P1, damped sys. task2
plt.plot(frequency, P1_resp_z_undamped_modal,label='modal estimation')
plt.plot(frequency, P1_resp_z_undamped[:,0],label='tf')
plt.title('Bodediagramm: undamped')
plt.ylabel('Amplitude (dB)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.legend()
plt.show()
```



[49]:
```
# Use W and V from previous calc
k = 2
W,V = eigsh(Kc,k,Mc,sigma=0,which='LM',maxiter = 1000) # takes way to long to␣
↪find samllest values

f_hat = np.zeros(3*N)
```

```python
f_hat[Iz[N1]] = 1.0    #for sys without constrains and force acting on N1 which␣
↪is the closest node to P1
fc_hat = f_hat[~Ic]    #for reduced sys, because of constrains

# Steps for calc
Nr_steps = 150.
steps = (300.-2.)/Nr_steps

P1_resp_z_undamped_modal_2 = np.array([])

for i in range(2, 300, round(steps)):
    omega = 2*np.pi*i
    H = receptanceMatrix(V,W,omega)
    x_hat = H @ fc_hat

    # insert missing nodes with zero
    x_hat_all = np.zeros(N*3) + complex(0,0)
    x_hat_all[~Ic] = x_hat

    P1_resp_z_undamped_modal_2 = np.concatenate( ( P1_resp_z_undamped_modal_2,␣
↪20*np.log10(np.array([np.abs(x_hat_all[Iz[N1]])]))) )
```

[50]:
```python
#plot response in z for P1, damped sys. task2
plt.plot(frequency, P1_resp_z_undamped_modal_2,label='modal estimation')
plt.plot(frequency, P1_resp_z_undamped[:,0],label='tf')
plt.title('Bodediagramm: undamped')
plt.ylabel('Amplitude (dB)')
plt.xlabel('Frequency (rad/s)')
plt.xscale('log')
plt.xlim(1, 1000)
plt.grid(True)
plt.legend()
plt.show()
```


Bodediagramm: undamped
modal estimation
tf
−60
−80
−100
−120
−140
−160
Amplitude (dB)
10^0
10^1
10^2
10^3
Frequency (rad/s)