Exercise 3

In this exercise you should invetigate model order reduction by a modeal basis. You should be able to re-use many parts from the previous exercises.

Consider the plate clamped at all edges.

In [1]:

```
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 vid
eo 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)
# np.set_printoptions(threshold=20)

from numpy.fft import rfft, rfftfreq

from utility_functions import Newmark
```

In [2]:

```
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)])

tol = 1e-12

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

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

# special points and the associated nodes
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))

# all node on the top of the plate
Nt = np.argwhere(np.abs(X[:,2]-X[:,2].max())<tol).ravel()

# 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
```

Constraint Enforcement

You can enforce contraints as in the previous exercises by selecting the appropriate rows from the system matrices, or use the nullsapce of the constraint matrix.

Set up a constraint matrix and use the provided function for computing the nullspace

```
from utlity_functions import nullspace
```

In [3]:

```
from utility_functions import nullspace
```

In [4]:

```
# 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)])
```

In [5]:

```
# B = np.zeros((len(If),3*N)) #Build constraints matrix
# B[np.arange(0,len(If)),np.sort(If)] = 1 #constraint the respective nodes
# Q = nullspace(B) #build the nullspace
```

In [6]:

```
# Calc constraint matrixes
# Q = csc_matrix(Q) #or make Q also a sparse and go with that
# K_bar = Q.transpose() @ K @ Q
# M_bar = Q.transpose() @ M @ Q
# C_bar = Q.transpose() @ C @ Q
```

Mode Shapes

Compute a set of mode-shapes of the system.

Note

We now use the sparse system because it's many times faster, however, one can just check the solution of the constraint system with the nullspace by enabling the code-lines and comparing it to the sparse solution.

In [7]:

```
# only compute a subset of modes of the reduced model
k = 10
Wc,Vc = eigsh(Kc,k,Mc,sigma=0,which='LM',maxiter = 1000)
```

In [8]:

```
# only compute a subset of modes of the reduced model
# k = 10
# W_bar,V_bar = eigsh(K_bar,k,M_bar,sigma=0,which='LM',maxiter = 1000)
```

Modal mass participation factor

Compute the modal mass participation factor for all 6 for the first 10 modes of the plate.

First you need to define the ridig body degrees of freedom (3 displacements and 3 rotations) in terms of displacement fields (can be seen as "mode shapes").

In [9]:

```
# W_unconstrained,V_unconstrained = eigsh(K,k,M,sigma=0,which='LM',maxiter = 1000)
```

In [10]:

```
def MPF(vi,M,ej) :
    return np.abs((vi @ M @ ej) / (vi @ M @ vi.transpose()))
    # return ((vi @ M @ ej) / (vi @ M @ vi.transpose()))
```

In [11]:

```
def plotMPF(ej, title) :
    dependency = np.zeros(k)
    for i,v in enumerate(Vc.T) :
        # dependency[i] = MPF(v, M, ej)
        dependency[i] = MPF(v, Mc, ej[~Ic])

    x = range(len(dependency))
    width = 0.75
    plt.bar(x, dependency, width, color="blue")
    plt.ylabel('Measure of dependency')
    plt.xlabel('Mode index')
    plt.title(title)
    plt.show()
```

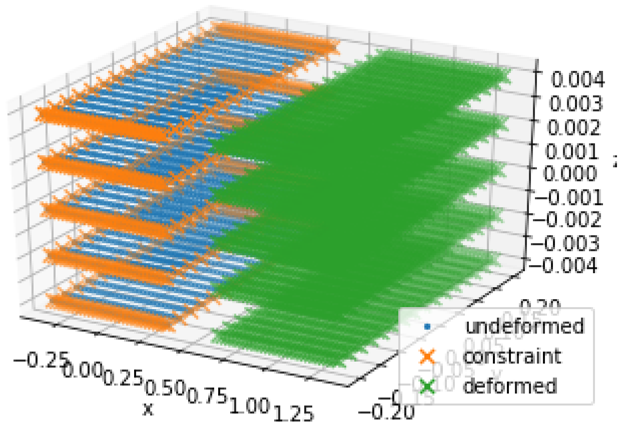
Then compute the 6 modal mass participation factors for each mode. Which ridid body displacement is most represented in which mode?

In [12]:

```
# Define rigid body displacements for
# X-DISPLACEMENT
e_x = X[:,0] + 1

e_x_all = np.zeros(3*N)
e_x_all[Ix] = e_x
e_x_all[Iy] = X[:,1]
e_x_all[Iz] = X[:,2]
e_x_all = e_x_all/np.linalg.norm(e_x_all)

fig,ax = plt.subplots(subplot_kw={'projection':'3d'})

# Plot it in 3D
ax.scatter(X[:,0],X[:,1],X[:,2],s=5,label='undeformed') # undeformed
ax.scatter(X[Nnosw,0],X[Nnosw,1],X[Nnosw,2],s=50,marker='x',label='constraint')
ax.scatter(e_x,X[:,1],X[:,2],s=50,marker='x',label='deformed')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend()
```

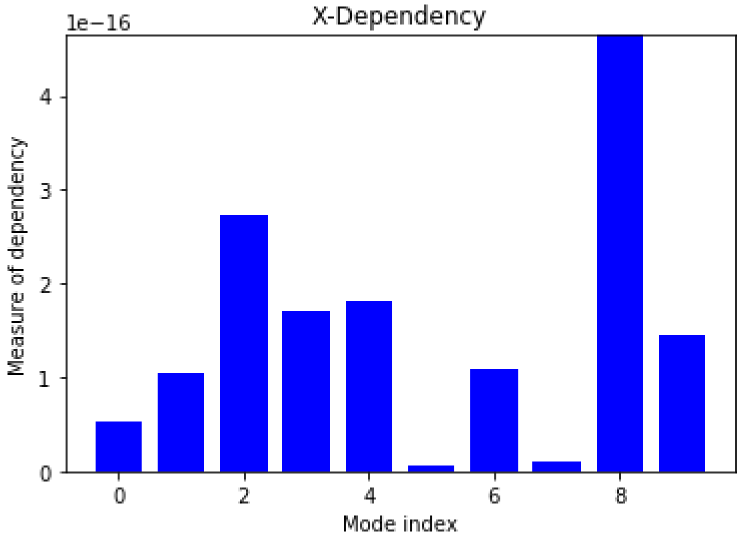
Out[12]:

```
<matplotlib.legend.Legend at 0x229a4171308>
```



In [13]:

```
plotMPF(e_x_all,"X-Dependency")
```

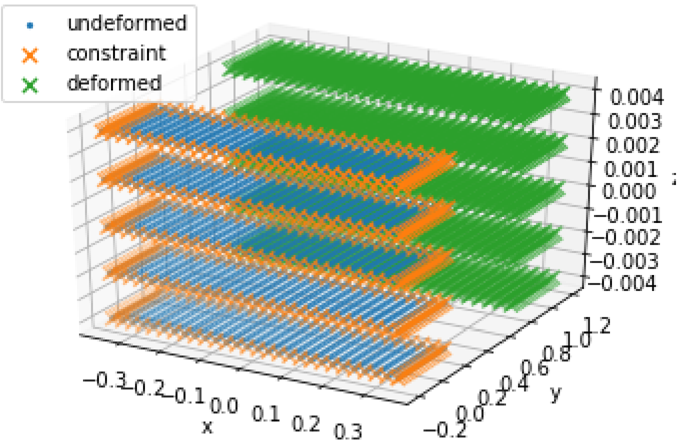


In [14]:

```
# Define rigid body displacements for
# Y-DISPLACEMENT
e_y = X[:,1] + 1

e_y_all = np.zeros(3*N)
e_y_all[Ix] = X[:,0]
e_y_all[Iy] = e_y
e_y_all[Iz] = X[:,2]
e_y_all = e_y_all/np.linalg.norm(e_y_all)

fig,ax = plt.subplots(subplot_kw={'projection':'3d'})

ax.scatter(X[:,0],X[:,1],X[:,2],s=5,label='undeformed') # undeformed
ax.scatter(X[Nnosw,0],X[Nnosw,1],X[Nnosw,2],s=50,marker='x',label='constraint')
ax.scatter(X[:,0],e_y,X[:,2],s=50,marker='x',label='deformed')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend()
```

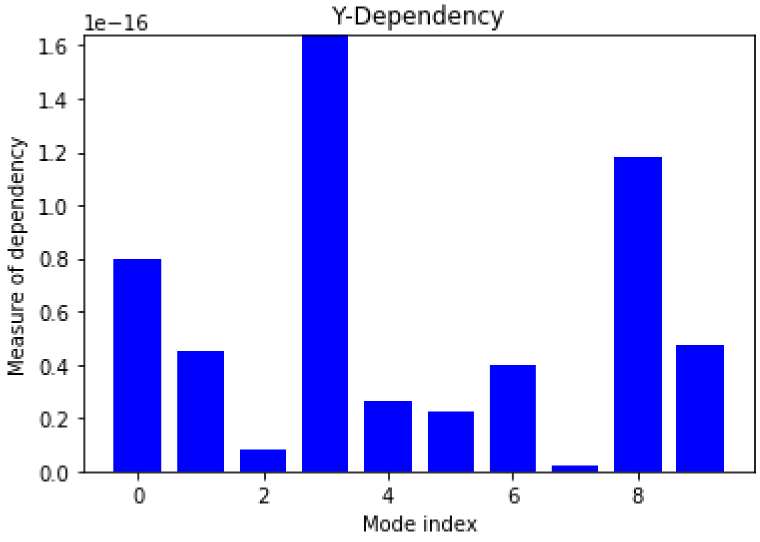
Out[14]:

```
<matplotlib.legend.Legend at 0x229a5131248>
```



In [15]:

```
plotMPF(e_y_all,"Y-Dependency")
```

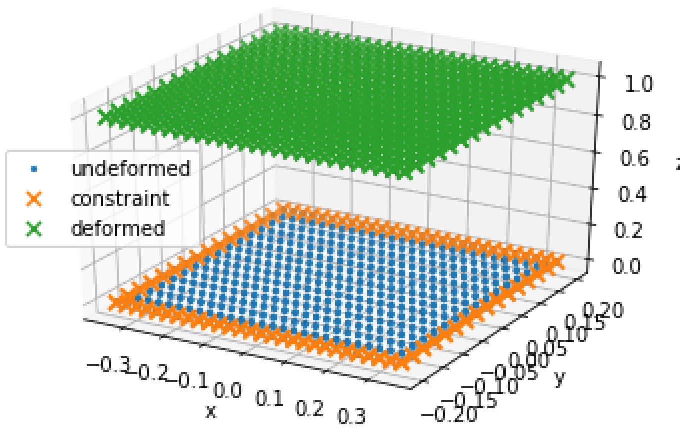


In [16]:

```
# Define rigid body displacements for
# Z-DISPLACEMENT
e_z = X[:,2] + 1

e_z_all = np.zeros(3*N)
e_z_all[Ix] = X[:,0]
e_z_all[Iy] = X[:,1]
e_z_all[Iz] = e_z
e_z_all = e_z_all/np.linalg.norm(e_z_all)

fig,ax = plt.subplots(subplot_kw={'projection':'3d'})

#Plot in 3D
ax.scatter(X[:,0],X[:,1],X[:,2],s=5,label='undeformed') # undeformed
ax.scatter(X[Nnosw,0],X[Nnosw,1],X[Nnosw,2],s=50,marker='x',label='constraint')
ax.scatter(X[:,0],X[:,1],e_z,s=50,marker='x',label='deformed')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend()
```

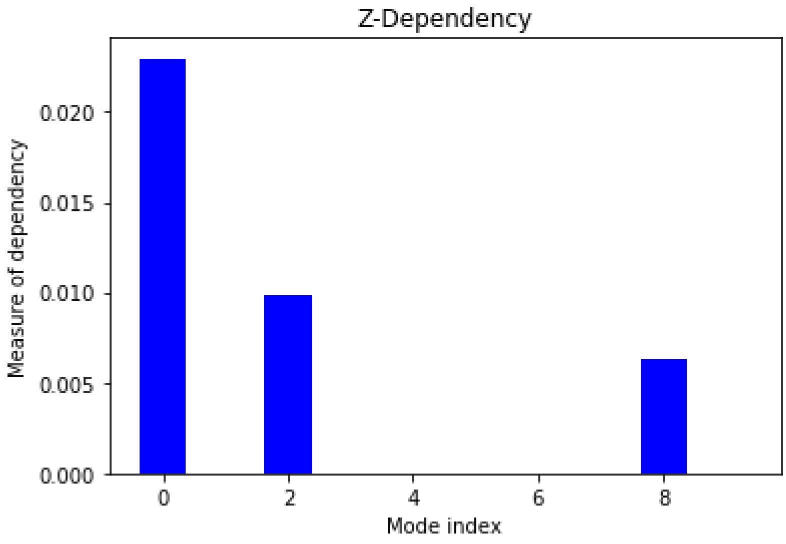
Out[16]:

```
<matplotlib.legend.Legend at 0x229a62d0b88>
```



In [17]:

```
plotMPF(e_z_all,"Z-Dependency")
```



In [18]:

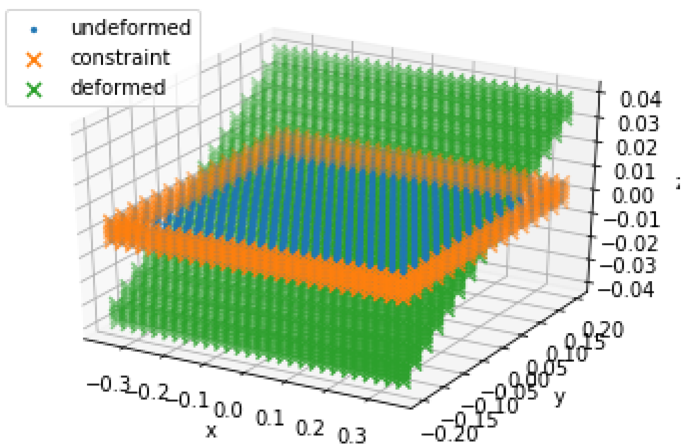
```
#Build rotation matrix
phi = 10*np.pi/180
Rx = np.array(((1,0,0),(0,np.cos(phi),-np.sin(phi)),(0,np.sin(phi),np.cos(phi))))
Ry = np.array(((np.cos(phi),0,np.sin(phi)),(0,1,0),(-np.sin(phi),0,np.cos(phi))))
Rz = np.array(((np.cos(phi),-np.sin(phi),0),(np.sin(phi),np.cos(phi),0),(0,0,1)))
```

In [19]:

```
# Rotation arround x-Axis
X_tran = X.transpose()
X_rot_tran = Rx @ X_tran
X_rot = X_rot_tran.transpose()

e_x_all = np.zeros(3*N)
e_x_all[Ix] = X_rot[:,0]
e_x_all[Iy] = X_rot[:,1]
e_x_all[Iz] = X_rot[:,2]
e_x_all = e_x_all/np.linalg.norm(e_x_all)

fig,ax = plt.subplots(subplot_kw={'projection':'3d'})

#Plot in 3D
ax.scatter(X[:,0],X[:,1],X[:,2],s=5,label='undeformed') # undeformed
ax.scatter(X[Nnosw,0],X[Nnosw,1],X[Nnosw,2],s=50,marker='x',label='constraint')
ax.scatter(X_rot[:,0],X_rot[:,1],X_rot[:,2],s=50,marker='x',label='deformed')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend()
```

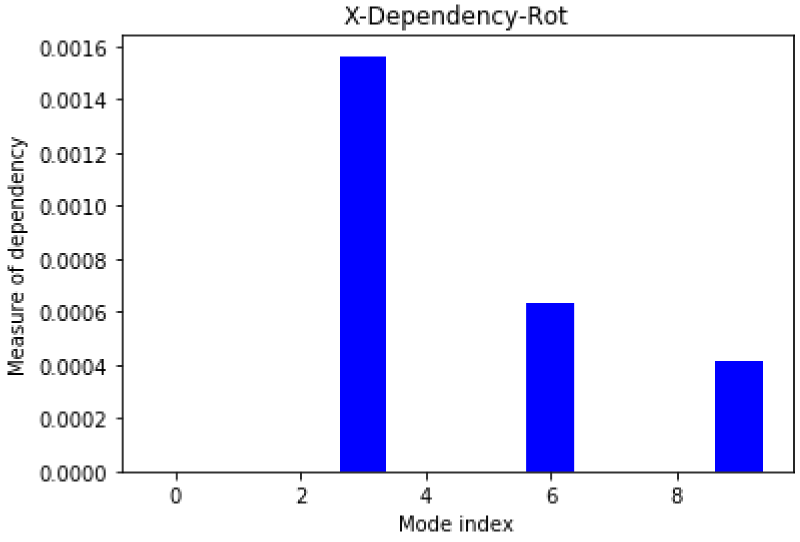
Out[19]:

```
<matplotlib.legend.Legend at 0x229a63a3888>
```



In [20]:

```
plotMPF(e_x_all,"X-Dependency-Rot")
```

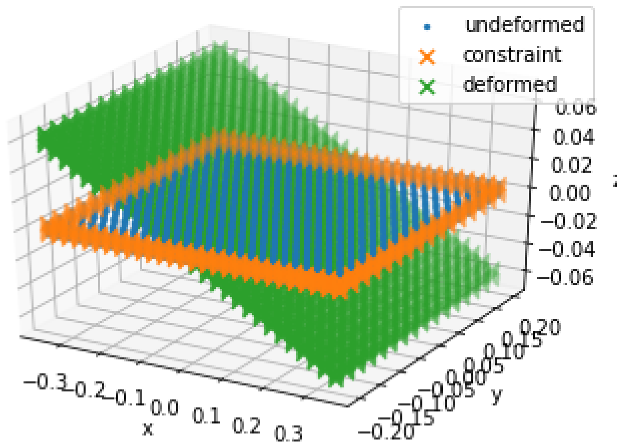


In [21]:

```
# Rotation arround y-Axis
X_tran = X.transpose()
X_rot_tran = Ry @ X_tran
X_rot = X_rot_tran.transpose()

e_y_all = np.zeros(3*N)
e_y_all[Ix] = X_rot[:,0]
e_y_all[Iy] = X_rot[:,1]
e_y_all[Iz] = X_rot[:,2]
e_y_all = e_y_all/np.linalg.norm(e_y_all)

fig,ax = plt.subplots(subplot_kw={'projection':'3d'})

#Plot in 3D
ax.scatter(X[:,0],X[:,1],X[:,2],s=5,label='undeformed') # undeformed
ax.scatter(X[Nnosw,0],X[Nnosw,1],X[Nnosw,2],s=50,marker='x',label='constraint')
ax.scatter(X_rot[:,0],X_rot[:,1],X_rot[:,2],s=50,marker='x',label='deformed')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend()
```

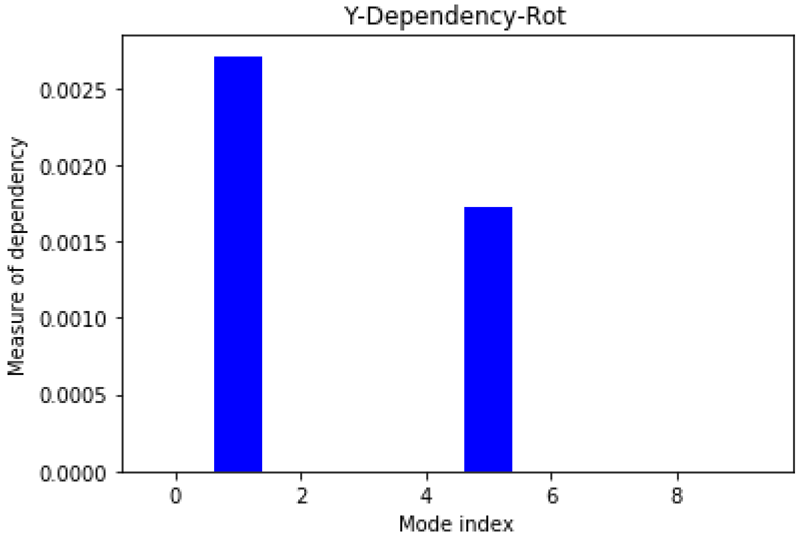
Out[21]:

```
<matplotlib.legend.Legend at 0x229a62e4448>
```



In [22]:

```
plotMPF(e_y_all,"Y-Dependency-Rot")
```

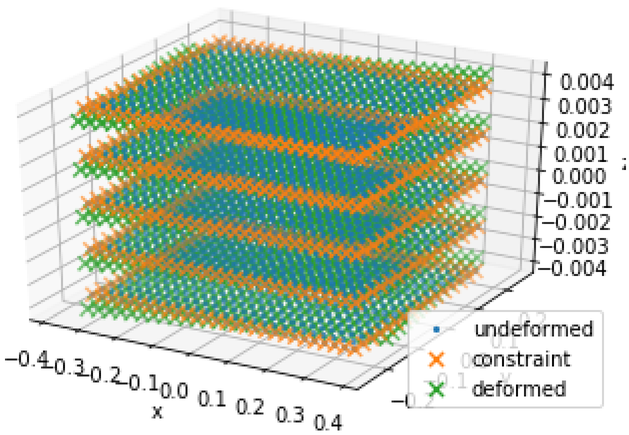


In [23]:

```
# Rotation arround z-Axis
X_tran = X.transpose()
X_rot_tran = Rz @ X_tran
X_rot = X_rot_tran.transpose()

e_z_all = np.zeros(3*N)
e_z_all[Ix] = X_rot[:,0]
e_z_all[Iy] = X_rot[:,1]
e_z_all[Iz] = X_rot[:,2]
e_z_all = e_z_all/np.linalg.norm(e_z_all)

fig,ax = plt.subplots(subplot_kw={'projection':'3d'})

#Plot in 3D
ax.scatter(X[:,0],X[:,1],X[:,2],s=5,label='undeformed') # undeformed
ax.scatter(X[Nnosw,0],X[Nnosw,1],X[Nnosw,2],s=50,marker='x',label='constraint')
ax.scatter(X_rot[:,0],X_rot[:,1],X_rot[:,2],s=50,marker='x',label='deformed')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend()
```

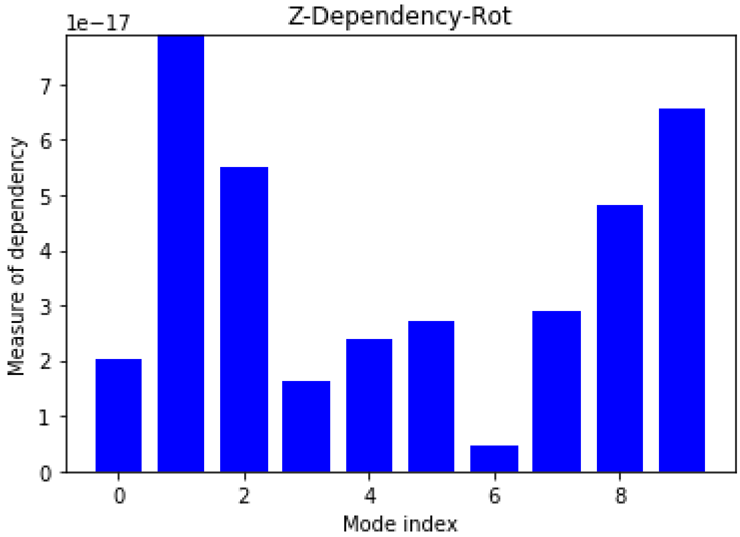
Out[23]:

```
<matplotlib.legend.Legend at 0x229a628c788>
```



In [24]:

```
plotMPF(e_z_all,"Z-Dependency-Rot")
```



Static Deformation

Check your bondary conditions by computing a static deformation: Assume a pressure acting on the plate (in transverse =z direction) which is linearly increasing from zero at one short edge (e.g. $x = x_{\min}$) to the oppsite edge. Assume a maximal pressure of 10kPa. For the sake of simplicity you can apply the pressure to one "node layer" (in thichness direction). Force per node can be obtained by multiplying by the "nodal area", i.e. the total area of the plate divided by the number of nodes in the "node layer".

In [25]:

```
# Node groups
tol = 1E-12
N_bot = np.argwhere(np.abs(X[:,2]-X[:,2].min())<tol).ravel() # Node indices of bottom n
odes
N_top = np.argwhere(np.abs(X[:,2]-X[:,2].max())<tol).ravel() # Node indices of top node
s

# Geometry
length_x = np.abs(x.max() - x.min())
length_y = np.abs(y.max() - y.min())
area = length_x * length_y
nodal_area = area / len(N_bot)

# Define loads (constant pressure for verification)
load_constant = np.zeros((3*N,1))
load_linear = np.zeros((3*N,1))
p_max = 10E3 # Max. pressure in Pa
p_mean = p_max / 2
load_constant[Iz[N_bot]] = p_mean * nodal_area


x_last = 0
for x_current in x: # Iterate over x-coordinates
    step_width = abs(x_current - x_last)
    nodes_current = np.argwhere(np.abs(X[N_bot][:,0] - x_current) < 0.25*step_width).ra
vel() # Look for according nodes
    pressure = p_max * (x_current - x[0]) / length_x # Assign linear pressure (Note: ze
ro at X[N_bot,0].min())
    load_linear[Iz[N_bot][nodes_current]] = pressure * nodal_area # Assign load at the
 current nodes
    x_last = x_current
    print(f"p(x = {x_current:.3f}) = {pressure:.1f}, F = {pressure * nodal_area:.3f} N
 at nodes = {N_bot[nodes_current]}")


# Constrain static system
from utility_functions import nullspace
B = np.zeros((len(If),3*N))
B[np.arange(0,len(If)),np.sort(If)] = 1 #constraint the respective nodes
Q = nullspace(B) #build the nullspace

Q = csc_matrix(Q)
Kc = csc_matrix(Q.T @ K @ Q)
load_constant_constrained = csc_matrix(Q.T @ load_constant)
load_linear_constrained = csc_matrix(Q.T @ load_linear)
```

```
p(x = -0.350) = 0.0, F = 0.000 N at nodes = [   0    5   10   15  140  145
150  155  280  285  290  295  420  425
  430  435  560  565  570  575  700  705  710  715  840  845  850  855
  980  985  990  995 1120 1125 1130 1135 1260 1265 1270 1275 1400 1405
 1410 1415 1540 1545 1550 1555 1680 1685 1690 1695 1820 1825 1830 1835
 1960 1965 1970 1975 2100 2105 2110 2115]
p(x = -0.324) = 370.4, F = 0.231 N at nodes = [   5  145  285  425  565  7
05  845  985 1125 1265 1405 1545 1685 1825
 1965 2105]
p(x = -0.298) = 740.7, F = 0.463 N at nodes = [  10  150  290  430  570  7
10  850  990 1130 1270 1410 1550 1690 1830
 1970 2110]
p(x = -0.272) = 1111.1, F = 0.694 N at nodes = [  15  155  295  435  575
715  855  995 1135 1275 1415 1555 1695 1835
 1975 2115]
p(x = -0.246) = 1481.5, F = 0.926 N at nodes = [  20  160  300  440  580
720  860 1000 1140 1280 1420 1560 1700 1840
 1980 2120]
p(x = -0.220) = 1851.9, F = 1.157 N at nodes = [  25  165  305  445  585
725  865 1005 1145 1285 1425 1565 1705 1845
 1985 2125]
p(x = -0.194) = 2222.2, F = 1.389 N at nodes = [  30  170  310  450  590
730  870 1010 1150 1290 1430 1570 1710 1850
 1990 2130]
p(x = -0.169) = 2592.6, F = 1.620 N at nodes = [  35  175  315  455  595
735  875 1015 1155 1295 1435 1575 1715 1855
 1995 2135]
p(x = -0.143) = 2963.0, F = 1.852 N at nodes = [  40  180  320  460  600
740  880 1020 1160 1300 1440 1580 1720 1860
 2000 2140]
p(x = -0.117) = 3333.3, F = 2.083 N at nodes = [  45  185  325  465  605
745  885 1025 1165 1305 1445 1585 1725 1865
 2005 2145]
p(x = -0.091) = 3703.7, F = 2.315 N at nodes = [  50  190  330  470  610
750  890 1030 1170 1310 1450 1590 1730 1870
 2010 2150]
p(x = -0.065) = 4074.1, F = 2.546 N at nodes = [  55  195  335  475  615
755  895 1035 1175 1315 1455 1595 1735 1875
 2015 2155]
p(x = -0.039) = 4444.4, F = 2.778 N at nodes = [  60  200  340  480  620
760  900 1040 1180 1320 1460 1600 1740 1880
 2020 2160]
p(x = -0.013) = 4814.8, F = 3.009 N at nodes = [  65  205  345  485  625
765  905 1045 1185 1325 1465 1605 1745 1885
 2025 2165]
p(x = 0.013) = 5185.2, F = 3.241 N at nodes = [  70  210  350  490  630  7
70  910 1050 1190 1330 1470 1610 1750 1890
 2030 2170]
p(x = 0.039) = 5555.6, F = 3.472 N at nodes = [  75  215  355  495  635  7
75  915 1055 1195 1335 1475 1615 1755 1895
 2035 2175]
p(x = 0.065) = 5925.9, F = 3.704 N at nodes = [  80  220  360  500  640  7
80  920 1060 1200 1340 1480 1620 1760 1900
 2040 2180]
p(x = 0.091) = 6296.3, F = 3.935 N at nodes = [  85  225  365  505  645  7
85  925 1065 1205 1345 1485 1625 1765 1905
 2045 2185]
p(x = 0.117) = 6666.7, F = 4.167 N at nodes = [  90  230  370  510  650  7
90  930 1070 1210 1350 1490 1630 1770 1910
 2050 2190]
p(x = 0.143) = 7037.0, F = 4.398 N at nodes = [  95  235  375  515  655  7
```

```
95  935 1075 1215 1355 1495 1635 1775 1915
 2055 2195]
p(x = 0.169) = 7407.4, F = 4.630 N at nodes = [ 100  240  380  520  660  8
00  940 1080 1220 1360 1500 1640 1780 1920
 2060 2200]
p(x = 0.194) = 7777.8, F = 4.861 N at nodes = [ 105  245  385  525  665  8
05  945 1085 1225 1365 1505 1645 1785 1925
 2065 2205]
p(x = 0.220) = 8148.1, F = 5.093 N at nodes = [ 110  250  390  530  670  8
10  950 1090 1230 1370 1510 1650 1790 1930
 2070 2210]
p(x = 0.246) = 8518.5, F = 5.324 N at nodes = [ 115  255  395  535  675  8
15  955 1095 1235 1375 1515 1655 1795 1935
 2075 2215]
p(x = 0.272) = 8888.9, F = 5.556 N at nodes = [ 120  260  400  540  680  8
20  960 1100 1240 1380 1520 1660 1800 1940
 2080 2220]
p(x = 0.298) = 9259.3, F = 5.787 N at nodes = [ 125  265  405  545  685  8
25  965 1105 1245 1385 1525 1665 1805 1945
 2085 2225]
p(x = 0.324) = 9629.6, F = 6.019 N at nodes = [ 130  270  410  550  690  8
30  970 1110 1250 1390 1530 1670 1810 1950
 2090 2230]
p(x = 0.350) = 10000.0, F = 6.250 N at nodes = [ 135  275  415  555  695
835  975 1115 1255 1395 1535 1675 1815 1955
 2095 2235]
```

In [26]:

```
from scipy.sparse.linalg import spsolve
uc_constant = spsolve(Kc, load_constant_constrained)
u_constant = Q @ uc_constant
uc_linear = spsolve(Kc, load_linear_constrained)
u_linear = Q @ uc_linear
```

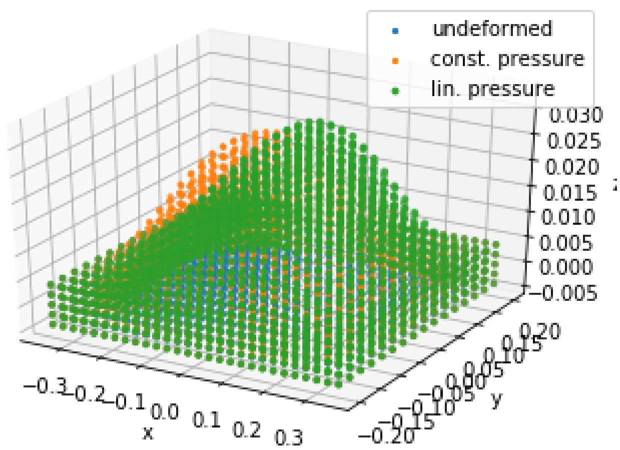
In [27]:

```
# Sanity check for load distribuation
print("LOAD CHARACTERISTIC:")
print(f"Total load: {load_linear.sum() :.1f} N")
print(f"Area: {area :.3f} m^2")
print(f"Calculated mean pressure: {load_linear.sum()/area/1000} kPa")
print(f"Analytic mean pressure: p_max/2 = {p_max/2/1000} kPa")

# Scatter plot
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_constant = np.array([u_constant[Ix],u_constant[Iy],u_constant[Iz]]).T
U_linear = np.array([u_linear[Ix],u_linear[Iy],u_linear[Iz]]).T

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

Xu_constant = X + s*U_constant # defomed configuration (displacement scaled by s)
Xu_linear = X + s*U_linear
ax.scatter(Xu_constant[:,0], Xu_constant[:,1], Xu_constant[:,2], s=5, label='const. pre
ssure')
ax.scatter(Xu_linear[:,0], Xu_linear[:,1], Xu_linear[:,2], s=8, label='lin. pressure')
#ax.scatter(Xu_diff[:,0], Xu_diff[:,1], Xu_diff[:,2], s=8, label='difference')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.legend()
```

```
LOAD CHARACTERISTIC:
Total load: 1400.0 N
Area: 0.280 m^2
Calculated mean pressure: 5.000000000000001 kPa
Analytic mean pressure: p_max/2 = 5.0 kPa
```

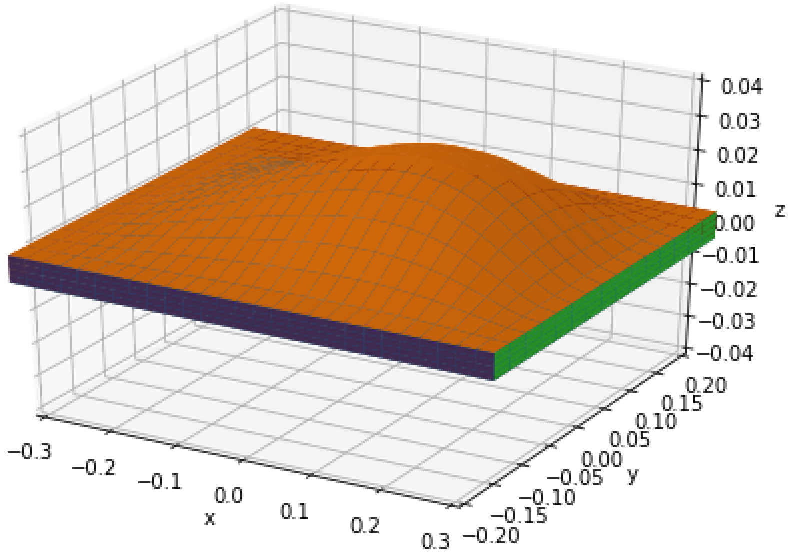
Out[27]:

```
<matplotlib.legend.Legend at 0x229a6583948>
```

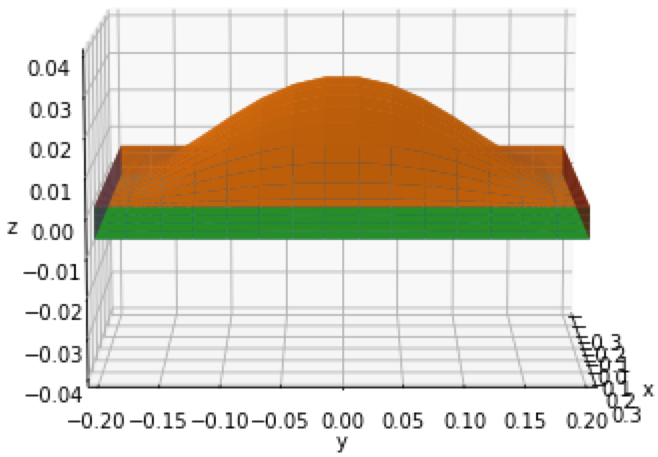


In [28]:

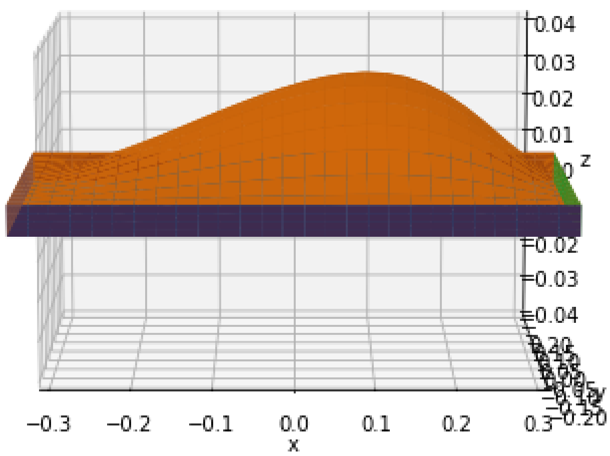
```
def plot_3d_deformation(X, u, elev=30.0, azim=60.0):

    # format U like X
    U = np.array([u[Ix],u[Iy],u[Iz]]).T
    s = 0.5/np.max(np.sqrt(np.sum(U**2,axis=0)))
    Xu = X + s*U

    # Set up figure
    fig = plt.figure()
    ax = Axes3D(fig, elev=elev, azim=azim)

    # Plot a basic wireframe.
    index = 1

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

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

    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')
    pass

plot_3d_deformation(X, u_linear, elev=30, azim=-60)
plot_3d_deformation(X, u_linear, elev=10, azim=0)
plot_3d_deformation(X, u_linear, elev=10, azim=-90)
```



0.04
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
−0.04
z
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
y
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x



0.04
0.03
0.02
0.01
z
0.00
−0.01
−0.02
−0.03
−0.04
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
y
x



0.04
0.03
0.02
0.01
z
−0.02
−0.03
−0.04
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
−0.20
y

Approximate the computed static displacement using the first three oscillation modes.

- What are the required modal coordinates?
- Plot the residual, which mode should you include to improve the approximation?

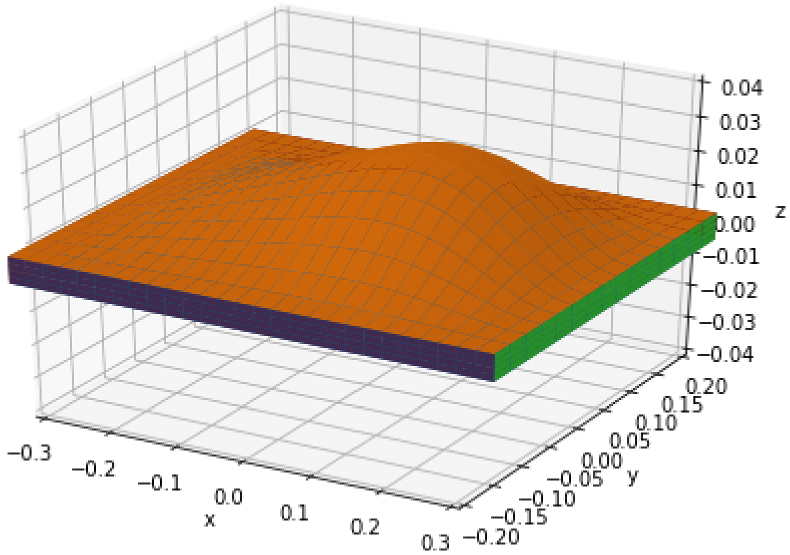
In [29]:

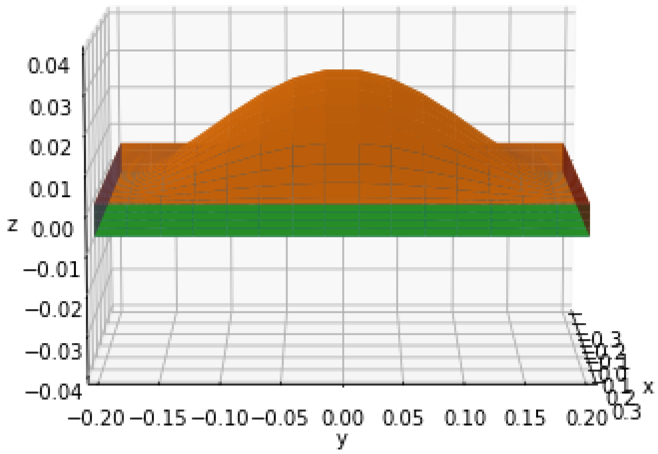
```
# Model order reduction
k = 3
Kc = Q.T @ K @ Q # Constrain system
Mc = Q.T @ M @ Q
fc = Q.T @ load_linear
Wc,Vc = eigsh(Kc,k,Mc,sigma=0,which='LM',maxiter = 1000) # Compute mode subset

Kc_reduced = Vc.T @ Kc @ Vc # Reduce system
fc_reduced = Vc.T @ fc

eta = spsolve(csc_matrix(Kc_reduced), csc_matrix(fc_reduced))
u_linear_reduced = Q @ Vc @ eta
```

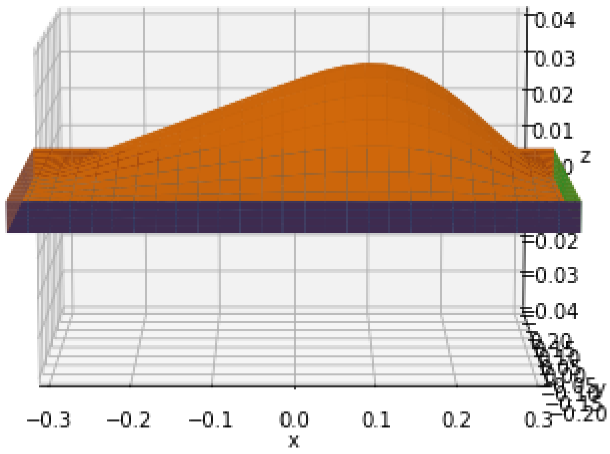
In [30]:

```
print("MODEL ORDER REDUCED SYSTEM:")
print(f"With {k} modes")
print("LOAD CHARACTERISTIC:")
print(f"Total load: {load_linear.sum() :.1f} N")
print(f"Calculated mean pressure: {load_linear.sum()/area/1000 :.1f} kPa")
plot_3d_deformation(X, u_linear_reduced, elev=30, azim=-60)
plot_3d_deformation(X, u_linear_reduced, elev=10, azim=0)
plot_3d_deformation(X, u_linear_reduced, elev=10, azim=-90)
```

```
MODEL ORDER REDUCED SYSTEM:
With 3 modes
LOAD CHARACTERISTIC:
Total load: 1400.0 N
Calculated mean pressure: 5.0 kPa
```

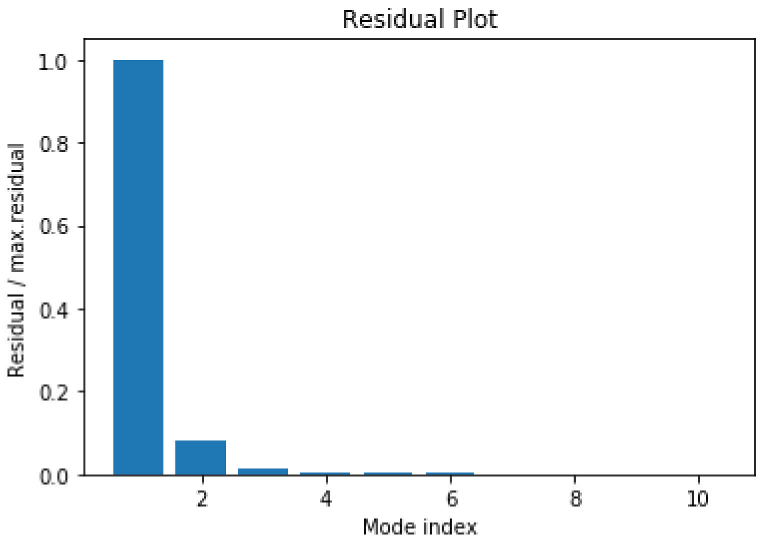






In [31]:

```
k = 10
Wc,Vc = eigsh(Kc,k,Mc,sigma=0,which='LM',maxiter = 1000) # Compute mode subset
residuals = []
for model_order in range(k):
    ls_solution, residual, rank, s = np.linalg.lstsq(Q @ Vc[:, 0:model_order], u_linear
, rcond=None)
    residuals.append(residual[0])

plt.bar(range(1,k+1), residuals / max(residuals))
plt.ylabel('Residual / max.residual')
plt.xlabel('Mode index')
plt.title('Residual Plot')
plt.show()
```



Transient Solution

We'll investigate the plate in the same configuration as in Exercise 2, but now compute results using reduced order models.

One can use the Newmark time intragration both for the full system and in the modal coodinates.

Forcing

Use the forcing given in Task 1 of Exercise 2: $f(t) = 1 - e^{-(t/0.002)^2}$ in z-direction at $P_1 = [0.2, 0.12, 0.003925]$.

In [32]:

```
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))

P_center = [0.,0.,0.]
N_center = np.argmin(np.sum((X-P_center)**2,axis=1))

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

## 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 ar
ray
    u[~Ic] = uc
    return u
```

In [33]:

```
def plot_P1_timedomain(u, time, N1):

    # Plot
    timePlot, timeAxis = plt.subplots(figsize=(20,8))
    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 plot_P2_timedomain(u, time, N2):

    # Plot
    timePlot, timeAxis = plt.subplots(figsize=(20,8))
    timeAxis.plot(time*1000, u[N2], label = "P2")
    timeAxis.set_xlabel('t [ms]')
    timeAxis.set_ylabel('u(t) [m]')
    timeAxis.set_title(f"Displacement - Time Domain")
    timeAxis.legend()

def compare_modalmodes(u_modal,u_full,time, N1, N2):

    # Plot
    timePlot_P1, timeAxis_P1 = plt.subplots(figsize=(20,8))
    timeAxis_P1.plot(time*1000, u_modal[N1], label = "P1_modal")
    timeAxis_P1.plot(time*1000, u_full[N1], label = "P1_full")

    timeAxis_P1.set_xlabel('t [ms]')
    timeAxis_P1.set_ylabel('u(t) [m]')
    timeAxis_P1.set_title(f"Displacement - Time Domain for P1")
    timeAxis_P1.legend()


    timePlot_P2, timeAxis_P2 = plt.subplots(figsize=(20,8))
    timeAxis_P2.plot(time*1000, u_modal[N2], label = "P2_modal")
    timeAxis_P2.plot(time*1000, u_full[N2], label = "P2_full")

    timeAxis_P2.set_xlabel('t [ms]')
    timeAxis_P2.set_ylabel('u(t) [m]')
    timeAxis_P2.set_title(f"Displacement - Time Domain for P2")
    timeAxis_P2.legend()

def plot_modalcoordinates(uc,time):
    #plot
    timePlot, timeAxis = plt.subplots(figsize=(20,8))
    for i in range(0,len(uc),1):
        timeAxis.plot(time*1000, uc[i], label = "Mode %s" %(i+1))
    timeAxis.set_xlabel('t [ms]')
    timeAxis.set_ylabel('Madalcoordinates(t) [m]')
    timeAxis.set_title(f"Modalbase with %s modes" %(len(uc)))
    timeAxis.legend()
```

Damping

For the sake of simplicity assume Rayleigh damping with $\alpha = 2.15$ and $\beta = 3e - 5$.

In [34]:

```
#Define alpha & beta
alpha = 2.15
beta = 3e-5

def full_excitation_analysis(tau=0.002, T=0.2,
                             excitation_type='step',
                             display_animation=False, k = None):

    # 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)

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

    Cc = alpha*Mc + beta*Kc # Construct the proportional damping matrix with pre-determ
ined alpha and beta values.

    f = np.array(np.zeros((3*N, time.shape[0]))) # Initialize load vector array; Note t
hat 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.

    #Cheking if full system is used or modal system
    if k == None:

        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.

        return u, uc
    else:
        # only compute a subset of modes of the reduced model
        W,V = eigsh(Kc,k,Mc,sigma=0,which='LM',maxiter = 1000)

        #Compute modal system matrices
        Km = V.T @ Kc @ V
        Mm = V.T @ Mc @ V
        Cm = V.T @ Cc @ V
```

```
        fm = V.T @ fc

        #define displacement vector
        u0c = np.array(np.zeros(len(fm),))

        # Time Integration
        uc, vc, ac = Newmark(Mm, Cm ,Km , fm, time, u0c)

        full_solution = V @ uc

        u = unreduce_constrained(full_solution, Ic) # Collect the displacement constrai
nts in the unreduced displacement array.

        return u, uc
```

In [35]:

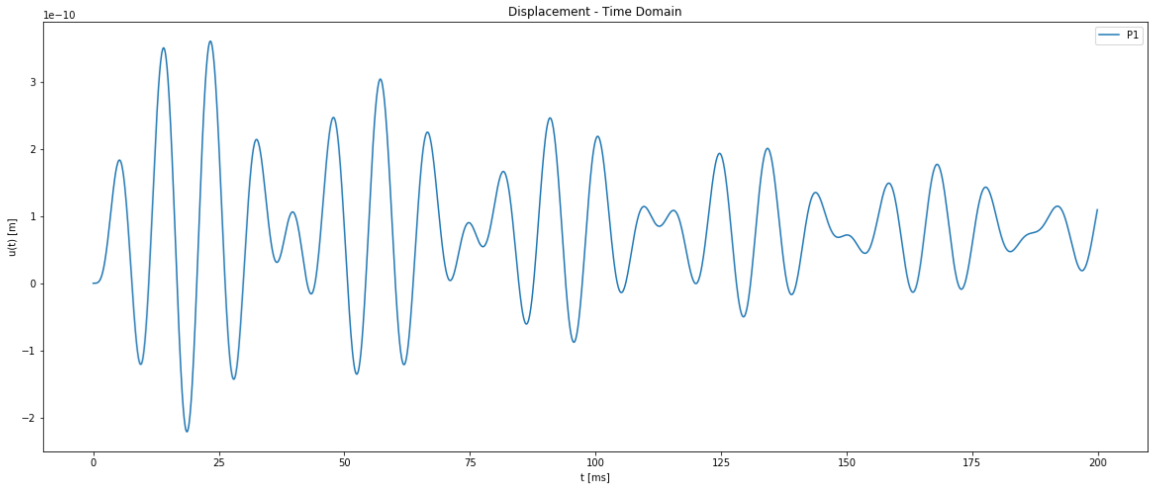
```
u, uc = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',k=2)
```

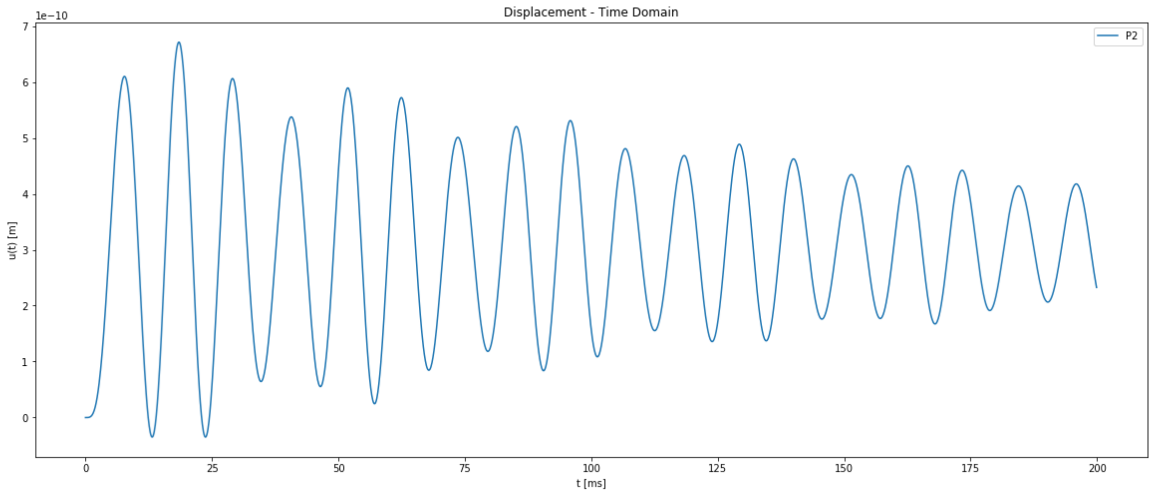
Task 1: Transient Response using Reduced Model

Use a modal basis of the first two modes and compute the transient response of the system (under the same loading as in Task 1 of Exercise 2). Plot the response at points P1 and P2, and compare with the full system. What is the error with respect to the full system?

In [36]:

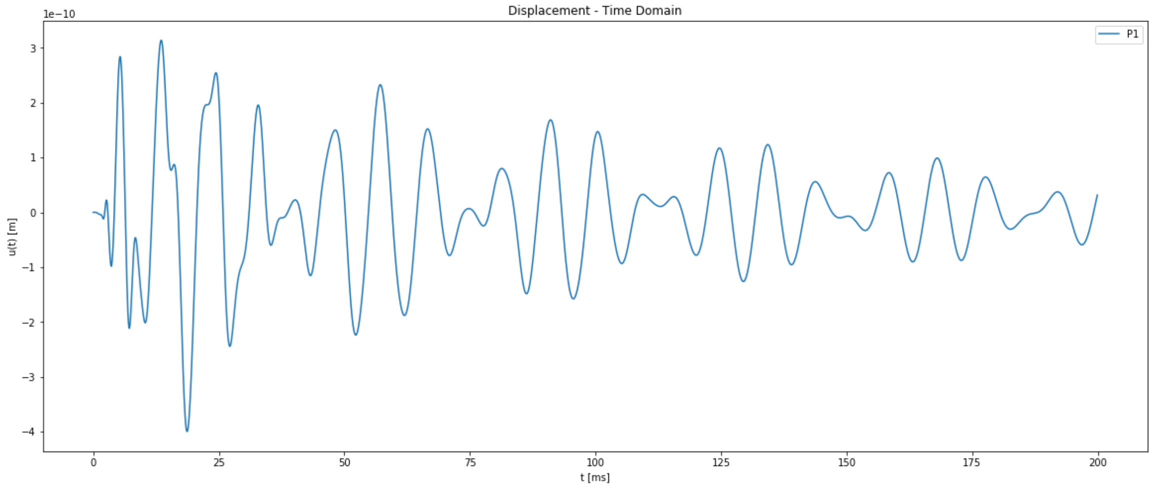
```
u_modal, uc_modal = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',k
=2)

f_max = 1/0.002 # Very crude estimation of max frequency.
dt = 1/(20*f_max) # Timestep
time = np.arange(0, 0.2, dt)

#plot P1 & P2 for 2 modes
plot_P1_timedomain(u_modal, time, N1)
plot_P2_timedomain(u_modal, time, N2)
```

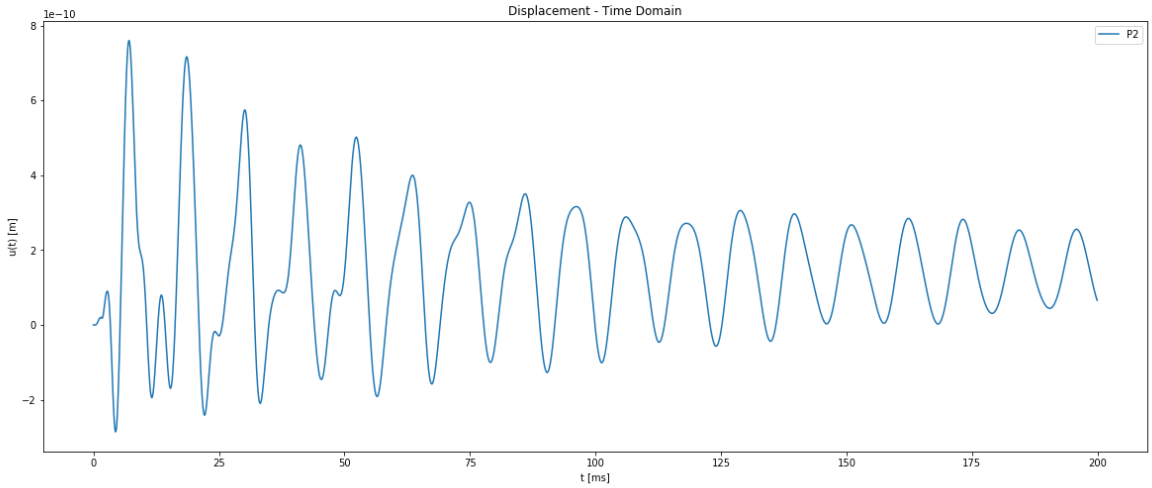




In [37]:

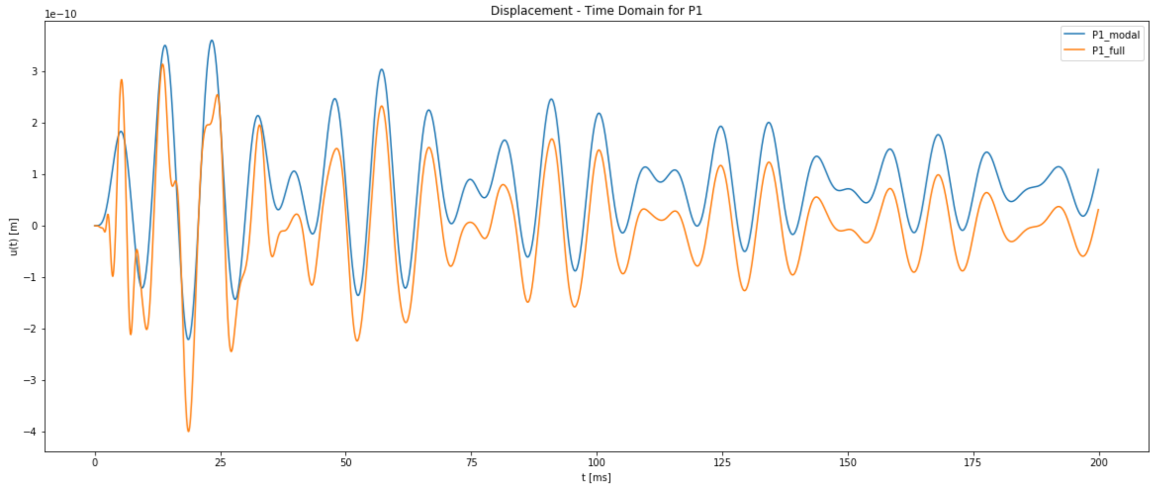
```
u_full, uc_full = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step')

#plot P1 & P2 for full system
plot_P1_timedomain(u_full, time, N1)
plot_P2_timedomain(u_full, time, N2)
```

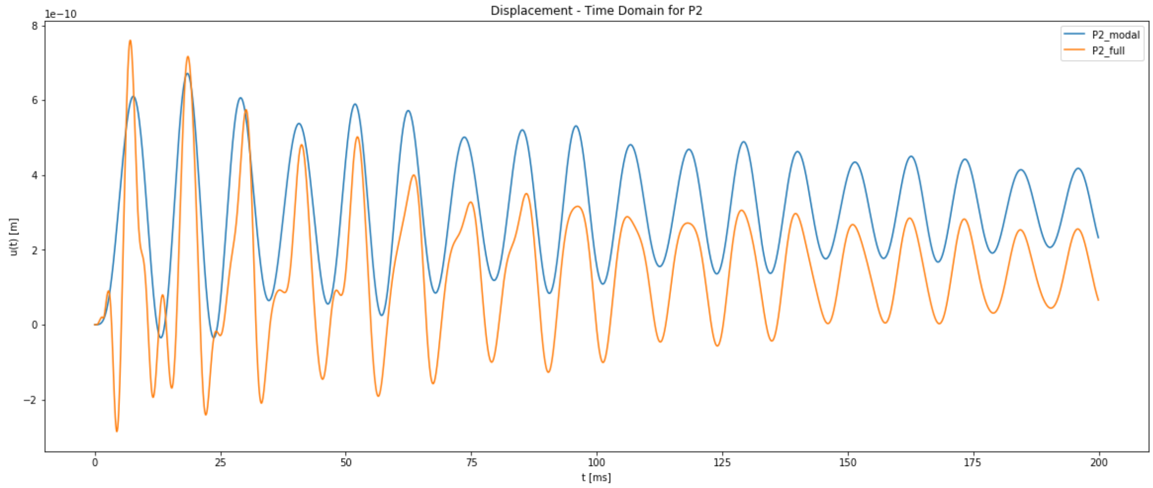




In [38]:

```
u_error_modal = u_full - u_modal

#Compare modalmodes with full system
compare_modalmodes(u_modal,u_full,time, N1, N2)
```





Choice of Modes

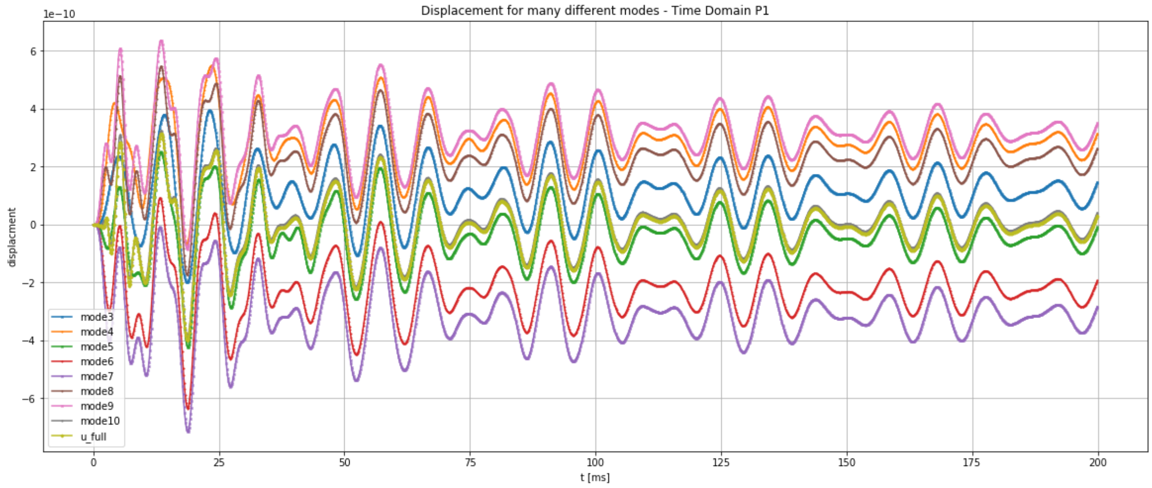
- How does the error improve when you take more modes?
- Plot the response at selected nodes, e.g. N1, N2, center, for different models in the same graph.

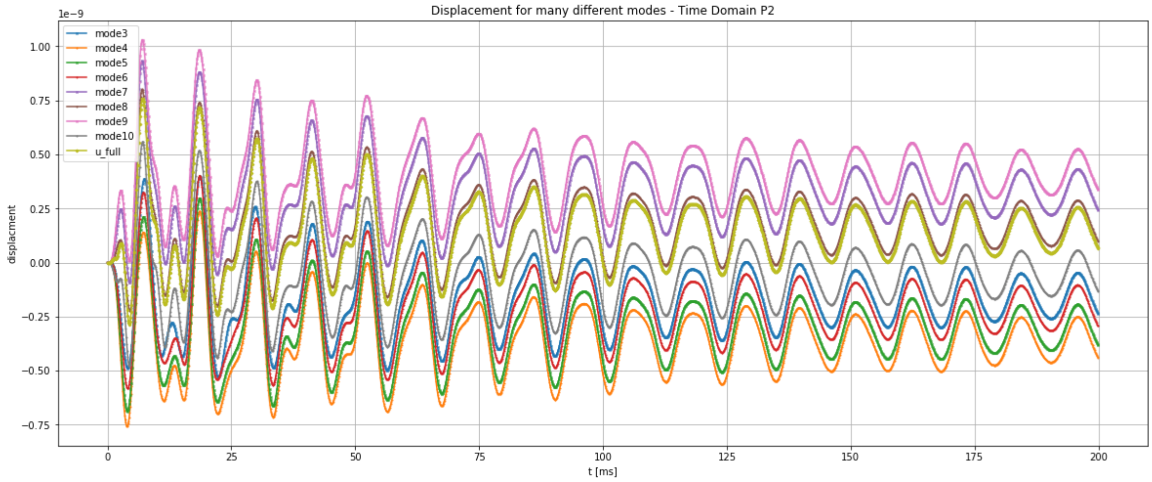
In [39]:

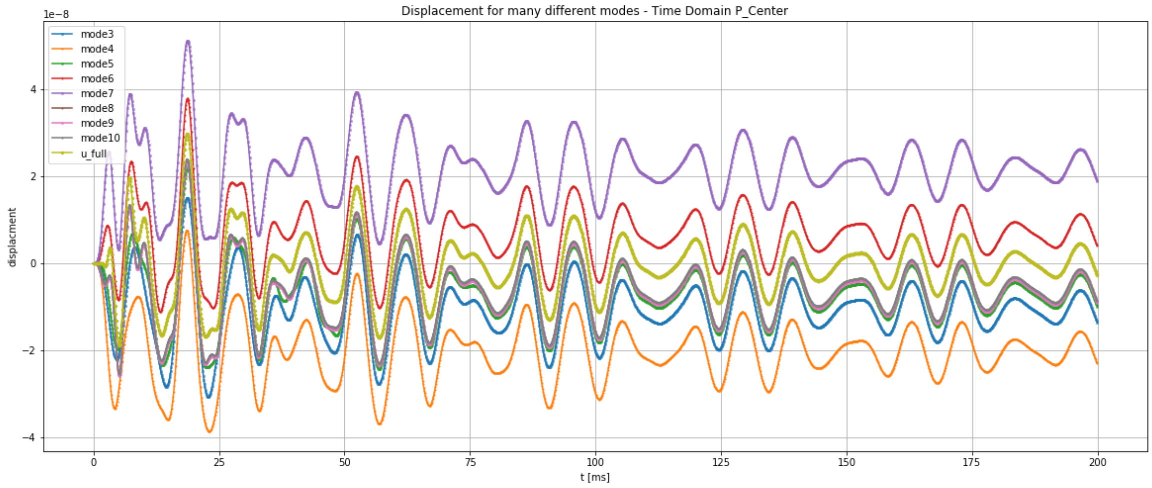
```
u_mode3, uc_mode3 = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',
k=3)
u_mode4, uc_mode4 = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',
k=4)
u_mode5, uc_mode5 = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',
k=5)
u_mode6, uc_mode6 = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',
k=6)
u_mode7, uc_mode7 = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',
k=7)
u_mode8, uc_mode8 = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',
k=8)
u_mode9, uc_mode9 = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step',
k=9)
u_mode10, uc_mode10 = full_excitation_analysis(tau=0.002, T=0.2, excitation_type='step'
, k=10)
```

In [40]:

```
u_modes = np.array((u_mode3,u_mode4,u_mode5,u_mode6,u_mode7,u_mode8,u_mode9,u_mode10,u_
full))
labels = ['mode3','mode4','mode5','mode6','mode7','mode8','mode9','mode10','u_full']
markers = ["x",".","x",".","x",".","x",".","D"]

fig, axP1 = plt.subplots(figsize=(20,8))
fig, axP2 = plt.subplots(figsize=(20,8))
fig, axP3 = plt.subplots(figsize=(20,8))

counter = 0
for i in u_modes:
    u = i
    axP1.plot(time*1000, i[N1],label = labels[counter], marker=markers[counter], marker
size=2) # plotting t, a separately
    axP2.plot(time*1000, i[N2],label = labels[counter], marker=markers[counter], marker
size=2)
    axP3.plot(time*1000, i[N_center],label = labels[counter], marker=markers[counter],
markersize=2)
    counter = counter + 1

axP1.set_xlabel('t [ms]')
axP1.set_ylabel('displacment')
axP1.set_title(f"Displacement for many different modes - Time Domain P1")
axP1.grid(True)
axP1.legend()

axP2.set_xlabel('t [ms]')
axP2.set_ylabel('displacment')
axP2.set_title(f"Displacement for many different modes - Time Domain P2")
axP2.grid(True)
axP2.legend()

axP3.set_xlabel('t [ms]')
axP3.set_ylabel('displacment')
axP3.set_title(f"Displacement for many different modes - Time Domain P_Center")
axP3.grid(True)
axP3.legend()

plt.show()
```

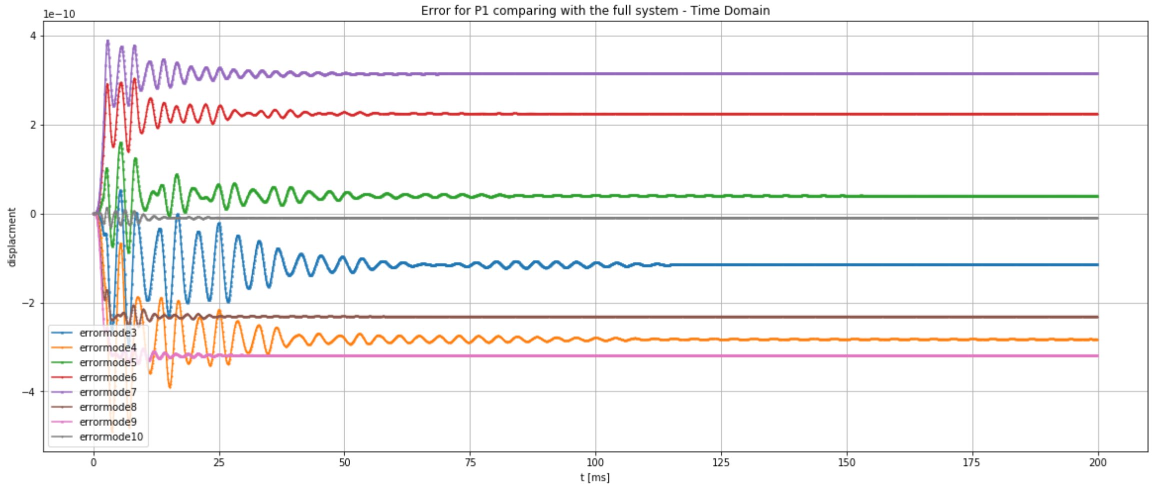

Displacement for many different modes - Time Domain P1
1e−10
displacment
t [ms]
mode3
mode4
mode5
mode6
mode7
mode8
mode9
mode10
u_full

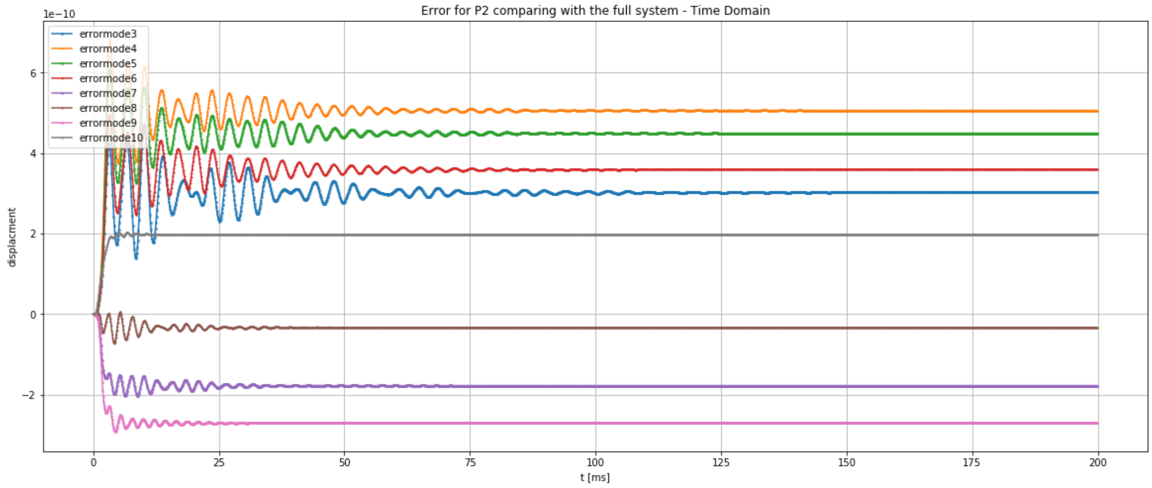

Displacement for many different modes - Time Domain P2
1e−9
displacment
t [ms]
mode3
mode4
mode5
mode6
mode7
mode8
mode9
mode10
u_full

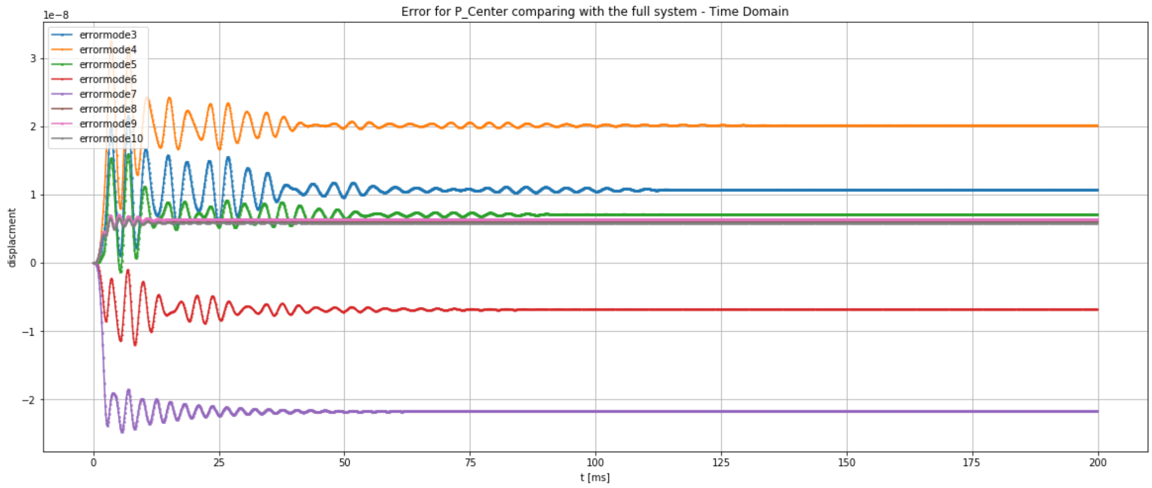

Displacement for many different modes - Time Domain P_Center
1e−8
displacment
t [ms]
mode3
mode4
mode5
mode6
mode7
mode8
mode9
mode10
u_full

In [41]:

```
#Comparing error
u_error_mode3 = u_full - u_mode3
u_error_mode4 = u_full - u_mode4
u_error_mode5 = u_full - u_mode5
u_error_mode6 = u_full - u_mode6
u_error_mode7 = u_full - u_mode7
u_error_mode8 = u_full - u_mode8
u_error_mode9 = u_full - u_mode9
u_error_mode10 = u_full - u_mode10

u_error_modes = np.array((u_error_mode3,u_error_mode4,u_error_mode5,u_error_mode6,u_err
or_mode7,
                          u_error_mode8,u_error_mode9,u_error_mode10))
labels = ['errormode3','errormode4','errormode5','errormode6','errormode7','errormode8'
,'errormode9','errormode10']
markers = ["x",".","x",".","x",".","x","."]

fig, axP1 = plt.subplots(figsize=(20,8))
fig, axP2 = plt.subplots(figsize=(20,8))
fig, axP3 = plt.subplots(figsize=(20,8))

counter = 0
for i in u_error_modes:
    u = i
    axP1.plot(time*1000, i[N1],label = labels[counter], marker=markers[counter], marker
size=2) # plotting t, a separately
    axP2.plot(time*1000, i[N2],label = labels[counter], marker=markers[counter], marker
size=2)
    axP3.plot(time*1000, i[N_center],label = labels[counter], marker=markers[counter],
markersize=2)
    counter = counter + 1

axP1.set_xlabel('t [ms]')
axP1.set_ylabel('displacment')
axP1.set_title(f"Error for P1 comparing with the full system - Time Domain")
axP1.grid(True)
axP1.legend()

axP2.set_xlabel('t [ms]')
axP2.set_ylabel('displacment')
axP2.set_title(f"Error for P2 comparing with the full system - Time Domain")
axP2.grid(True)
axP2.legend()

axP3.set_xlabel('t [ms]')
axP3.set_ylabel('displacment')
axP3.set_title(f"Error for P_Center comparing with the full system - Time Domain")
axP3.grid(True)
axP3.legend()

plt.show()
```


Error for P1 comparing with the full system - Time Domain
1e−10
displacment
t [ms]
errormode3
errormode4
errormode5
errormode6
errormode7
errormode8
errormode9
errormode10


Error for P2 comparing with the full system - Time Domain
1e−10
displacment
t [ms]
errormode3
errormode4
errormode5
errormode6
errormode7
errormode8
errormode9
errormode10

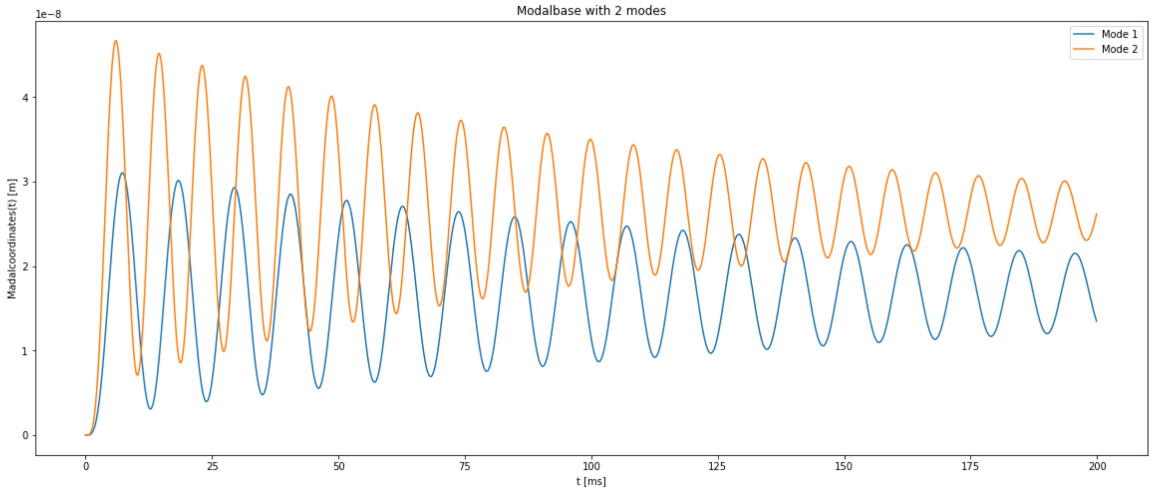

Error for P_Center comparing with the full system - Time Domain
1e−8
displacment
t [ms]
errormode3
errormode4
errormode5
errormode6
errormode7
errormode8
errormode9
errormode10

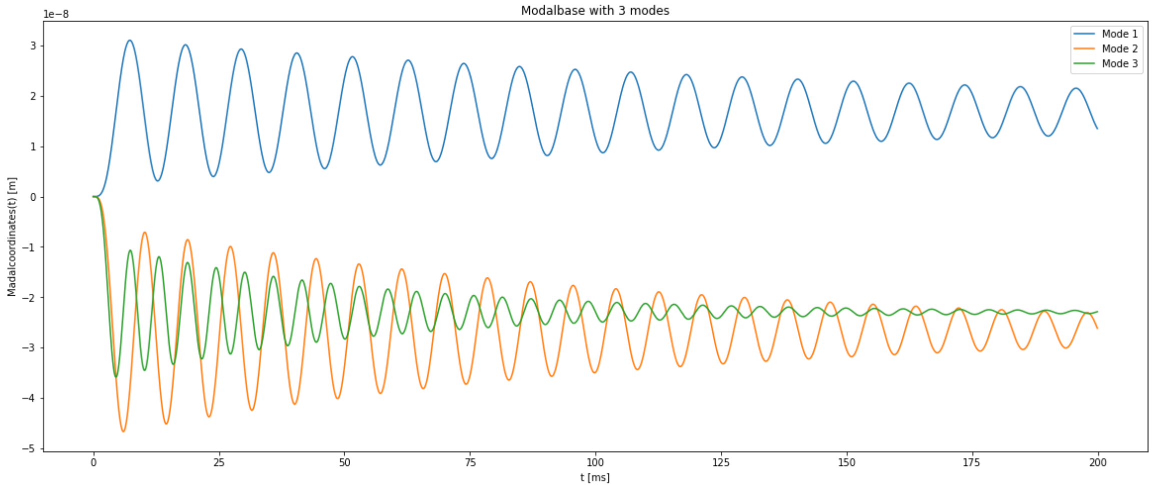
Time Evolution of Modal Corrdinates

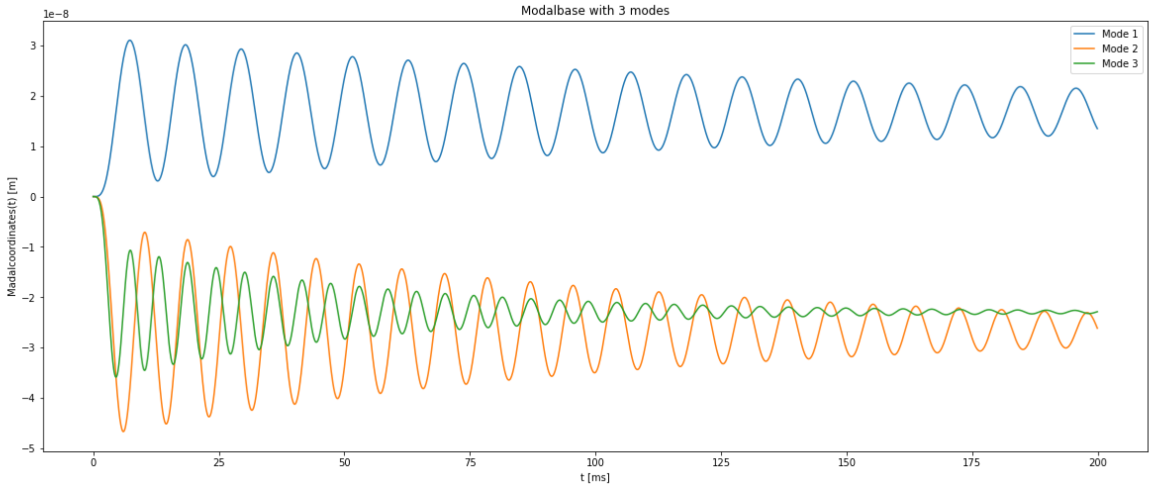
- Visualize the time evolution of the used modal coordinates
- Do this for the results obtained with differnt modal bases
- Compute the modal contributions in the same way. Which modes contribute most for which model?

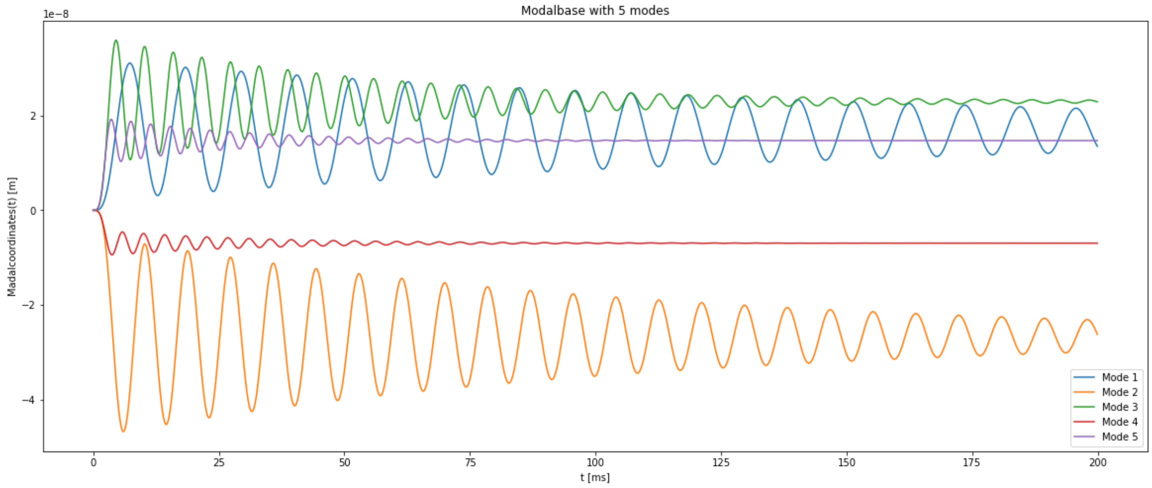
In [42]:

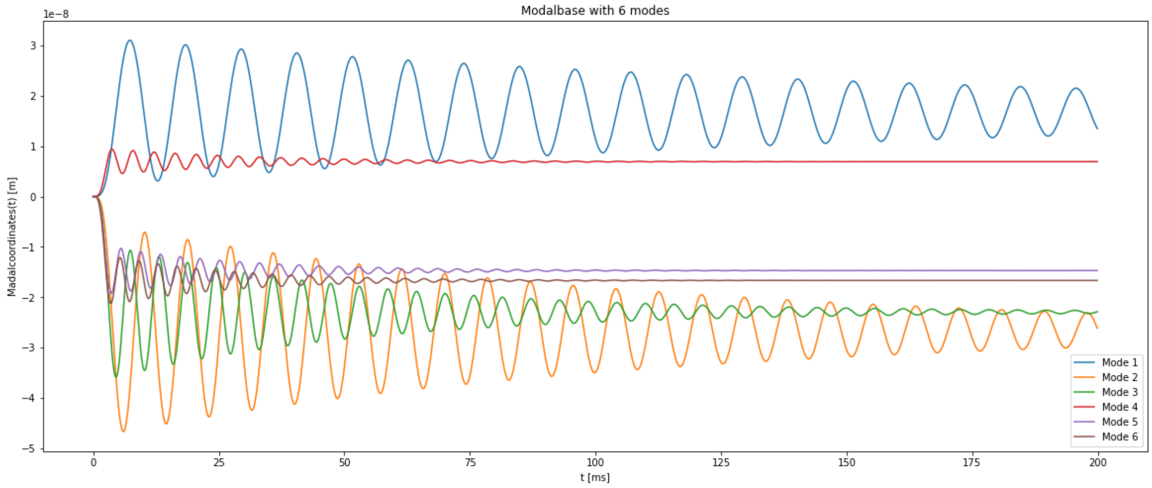
```
#Visualizing time evoultion of the modal cooridnates for different modal bsases
plot_modalcoordinates(uc_modal,time)
plot_modalcoordinates(uc_mode3,time)
plot_modalcoordinates(uc_mode3,time)
plot_modalcoordinates(uc_mode5,time)
plot_modalcoordinates(uc_mode6,time)
plot_modalcoordinates(uc_mode7,time)
plot_modalcoordinates(uc_mode8,time)
plot_modalcoordinates(uc_mode10,time)
```

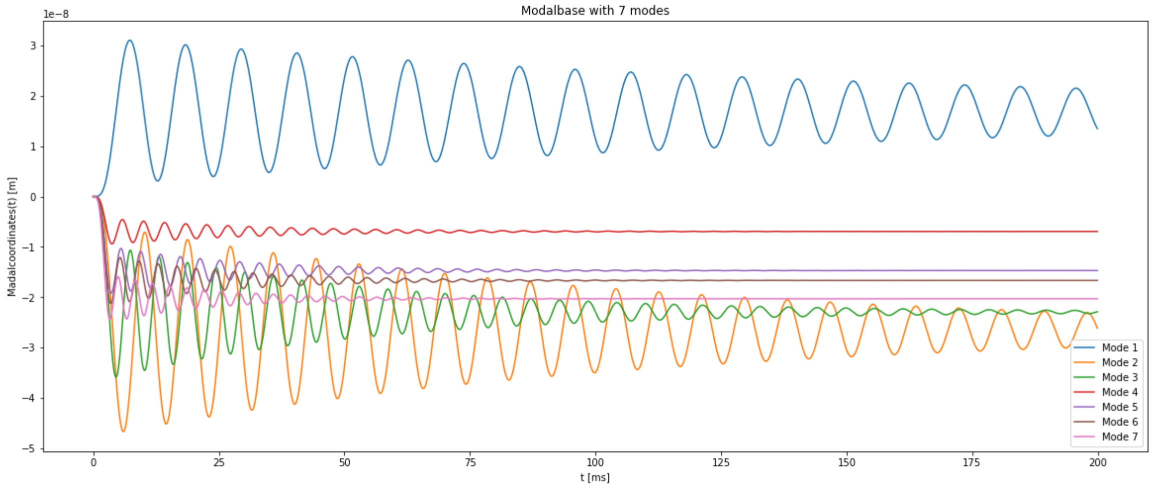

Modalbase with 2 modes
1e−8
Mode 1
Mode 2
Madalcoordinates(t) [m]
t [ms]

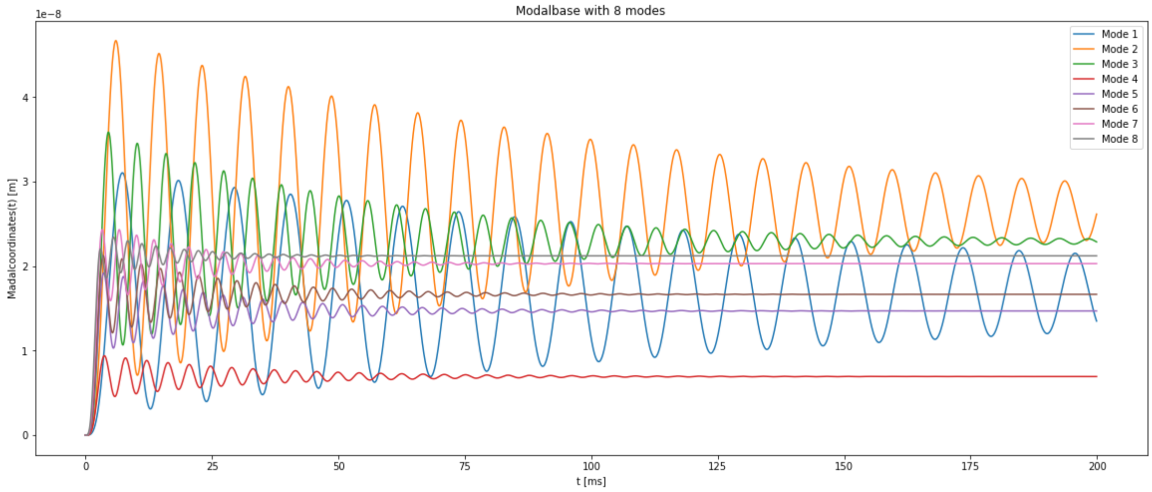

Modalbase with 3 modes
1e−8
Mode 1
Mode 2
Mode 3
Madalcoordinates(t) [m]
t [ms]

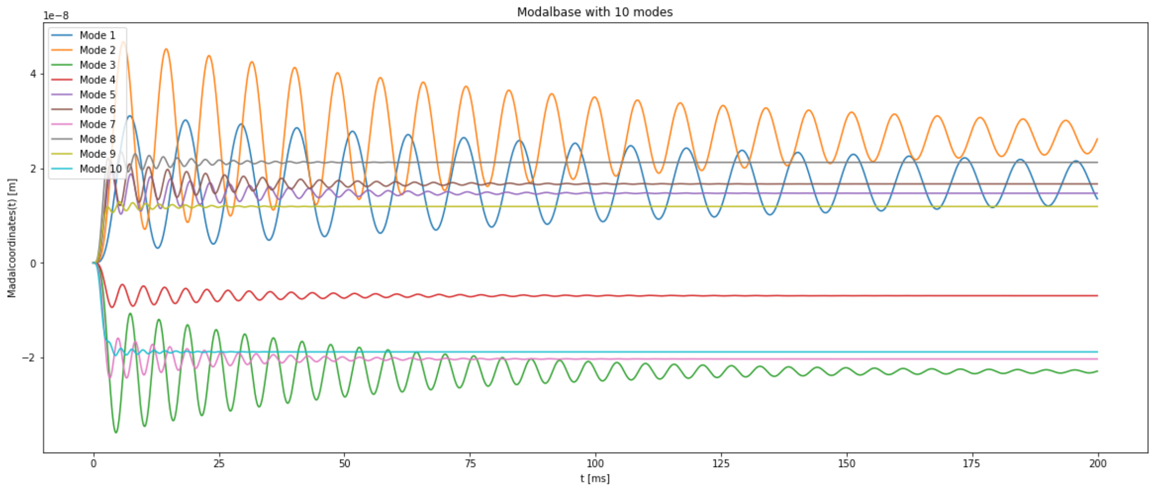

Modalbase with 3 modes
1e−8
Mode 1
Mode 2
Mode 3
Madalcoordinates(t) [m]
t [ms]


Modalbase with 5 modes
1e−8
Madalcoordinates(t) [m]
t [ms]
Mode 1
Mode 2
Mode 3
Mode 4
Mode 5


Modalbase with 6 modes
1e−8
Madalcoordinates(t) [m]
t [ms]
Mode 1
Mode 2
Mode 3
Mode 4
Mode 5
Mode 6


Modalbase with 7 modes
1e−8
Madalcoordinates(t) [m]
t [ms]
Mode 1
Mode 2
Mode 3
Mode 4
Mode 5
Mode 6
Mode 7



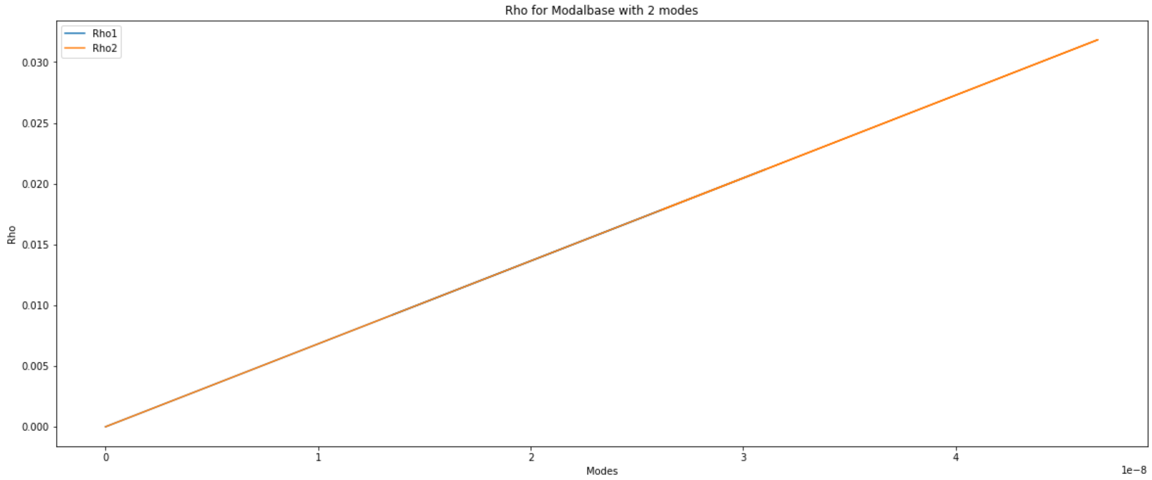


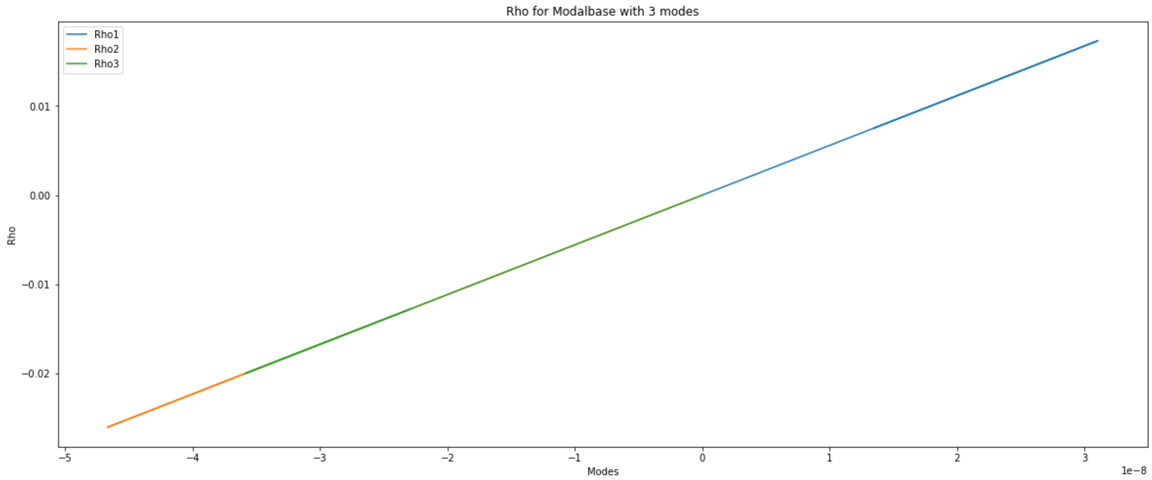
In [43]:

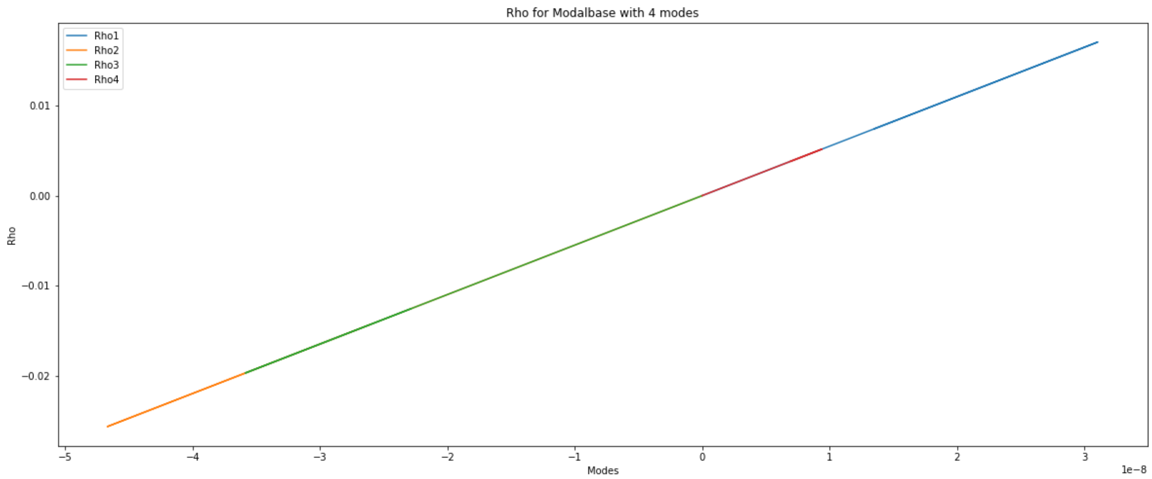
```
#Define a function that computes rho for diffferent modal cooridnates
def plot_rho(uc):
    rho = uc/np.linalg.norm(uc)
    Plot, Axis = plt.subplots(figsize=(20,8))
    for i in range(0,len(uc),1):
        Axis.plot(uc[i], rho[i] , label = "Rho%s" %(i+1))
    Axis.set_xlabel('Modes')
    Axis.set_ylabel('Rho')
    Axis.set_title(f"Rho for Modalbase with %s modes" %(len(uc)))
    Axis.legend()
```

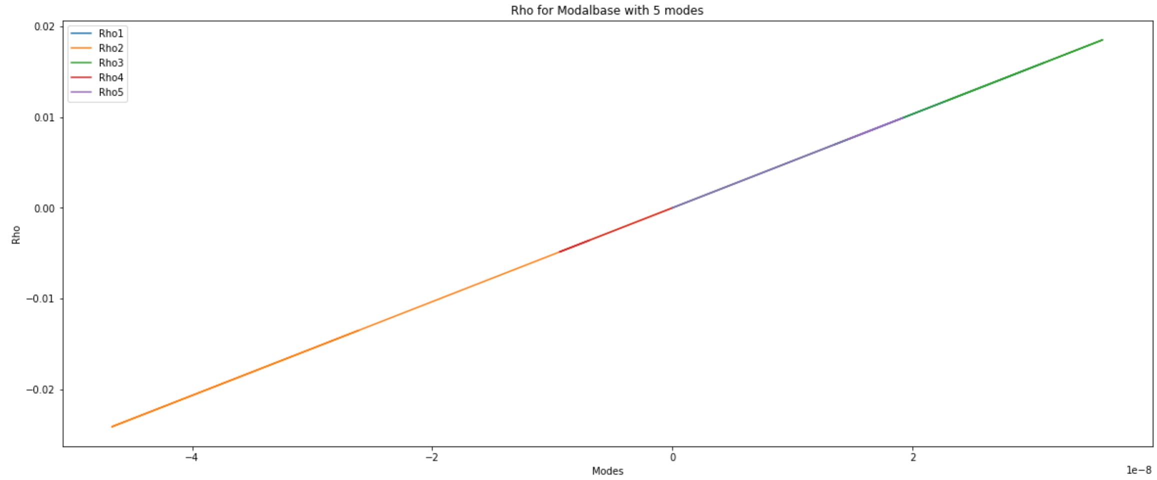
In [44]:

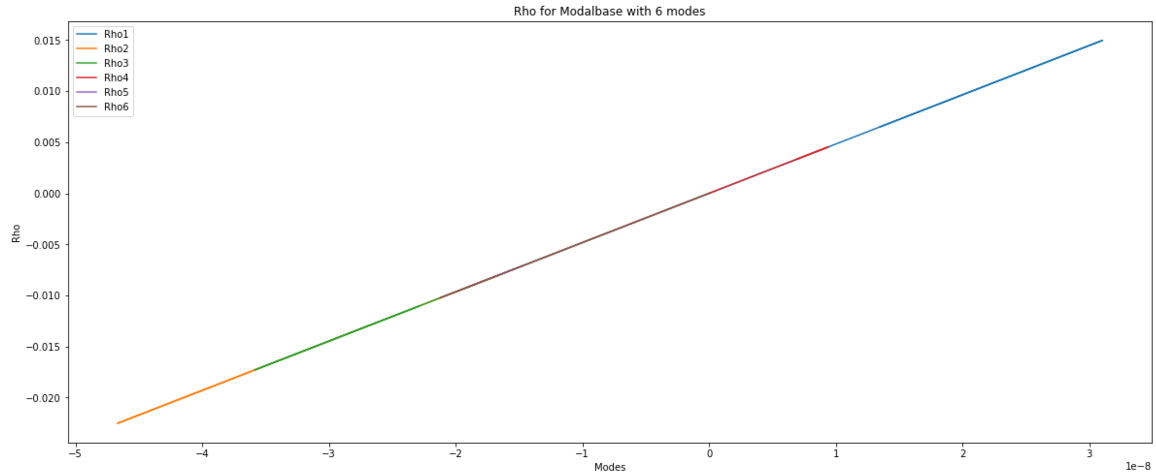
```
#Visualizing time evoultion of the modal cooridnates for different modal bsases
plot_rho(uc_modal)
plot_rho(uc_mode3)
plot_rho(uc_mode4)
plot_rho(uc_mode5)
plot_rho(uc_mode6)
plot_rho(uc_mode7)
plot_rho(uc_mode8)
plot_rho(uc_mode9)
plot_rho(uc_mode10)
```

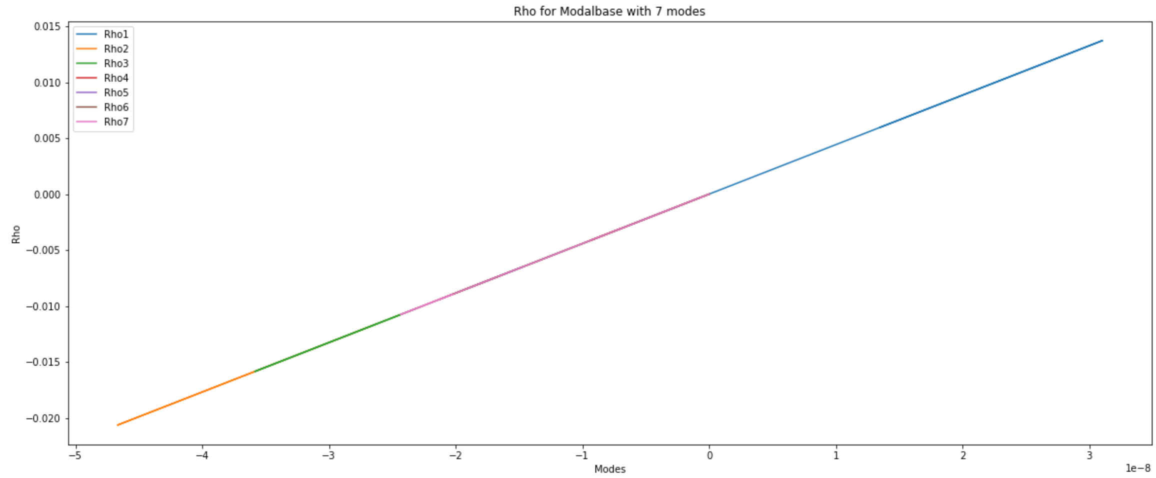

Rho for Modalbase with 2 modes
Rho1
Rho2
0.030
0.025
0.020
0.015
0.010
0.005
0.000
Rho
0
1
2
3
4
Modes
1e−8

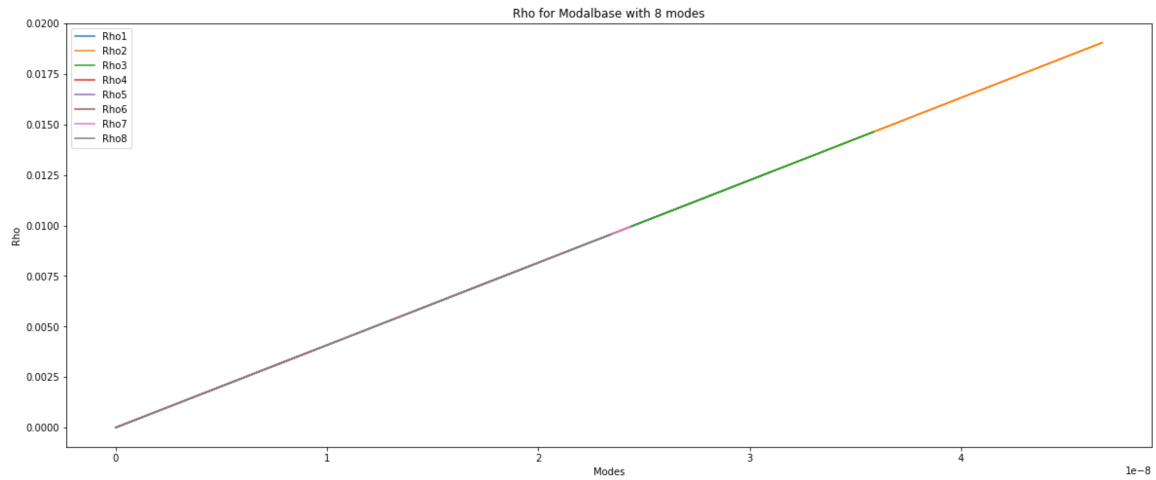

Rho for Modalbase with 3 modes
Rho1
Rho2
Rho3
0.01
0.00
−0.01
−0.02
Rho
−5
−4
−3
−2
−1
0
1
2
3
Modes
1e−8

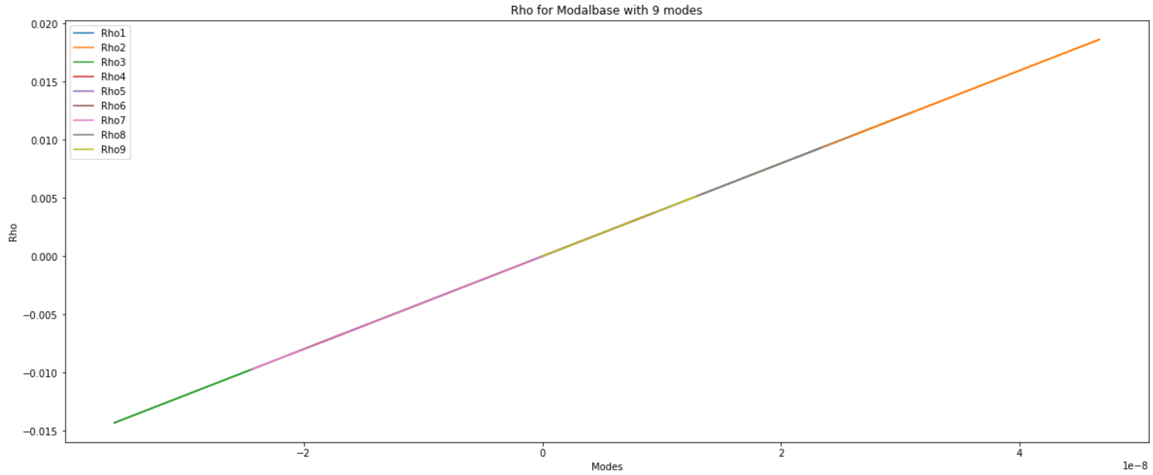

Rho for Modalbase with 4 modes
Rho1
Rho2
Rho3
Rho4
0.01
0.00
−0.01
−0.02
Rho
−5
−4
−3
−2
−1
0
1
2
3
Modes
1e−8

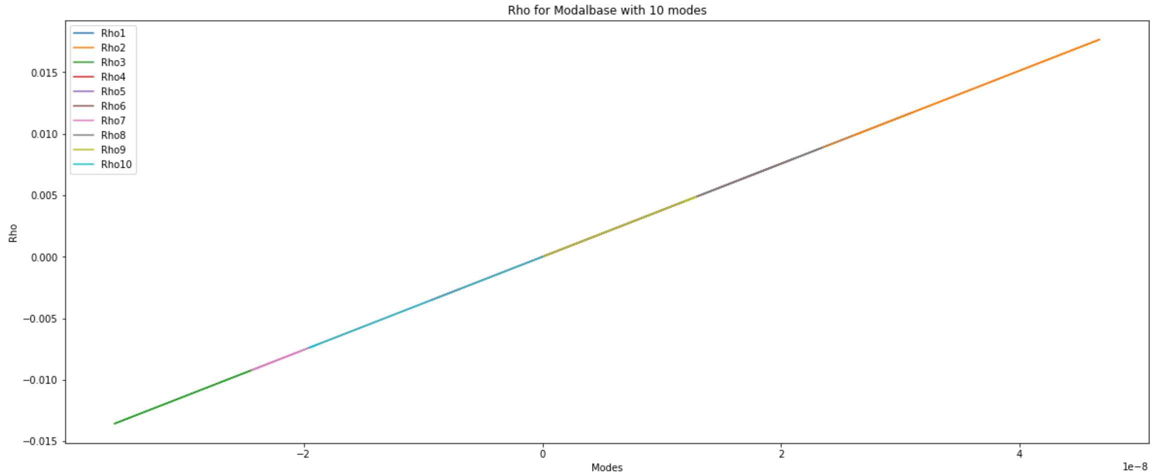

Rho for Modalbase with 5 modes
Rho1
Rho2
Rho3
Rho4
Rho5
Rho
Modes
1e−8


Rho for Modalbase with 6 modes
Rho1
Rho2
Rho3
Rho4
Rho5
Rho6
Rho
Modes
1e−8


Rho for Modalbase with 7 modes
Rho1
Rho2
Rho3
Rho4
Rho5
Rho6
Rho7
Rho
Modes
1e−8


Rho for Modalbase with 8 modes
Rho1
Rho2
Rho3
Rho4
Rho5
Rho6
Rho7
Rho8
Rho
Modes
1e−8





Steady State Oscillation | Frequency Domain

Now switch to frequency domain and compute the steady state response of the system. For the sake of simplicity use a unit excitation at P_1.

In [45]:

```
from numpy.linalg import solve
import time
```

In [46]:

```
def FrequencyDomain(omega, direc = Iz, node = N1, K = Kc, C = Cc, M = Mc):
    #1. Compute the dynamic stiffness matrix Z for one omega
    Z = K + complex(0,1) * omega * C - omega**2 * M

    #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 whic
h is the closest node to P1
    fc_hat = f_hat[~Ic]    #for reduced sys, because of constrains

    fc_hat_red = V.transpose() @ fc_hat #reduced forcing vector

    #3. solve for the displacements
    xc_hat_red = solve(Z,fc_hat_red)       #for np.array matrices

    return(xc_hat_red) #complex, so ampl and phase is in there; for all DoF which are n
ot constrained
```

Task 2: Compute Harmonic Response using a Reduced Model

Use the first 10 modes to compute the steady state response for a unit forcing in z-direction at P_1. Do the computation for Rayleigh damping and for Modal damping with a damping ratio of 0.01 for each mode. Compare the results by plotting the transfer functions up to 300Hz.

In [47]:

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

In [48]:

```
def ResponseOverReducedSystem(max_freq = 300, min_freq = 2, Nr_steps = 150):

    ## Compute the reduced system matrices and forcing vector
    M_red = V.transpose() @ Mc @ V
    C_red = V.transpose() @ Cc @ V
    K_red = V.transpose() @ Kc @ V

    ## Solve the reduced system for the modal coordinates eta and transforamtion to obt
ain the full solution
    eta_hat_store = []

    freq = np.linspace(min_freq, max_freq, Nr_steps)
    P1_resp_z = np.zeros([len(freq), 2])

    for i,f in enumerate(freq):
        # response of the reduced system M_red K_red C_red
        eta_hat = FrequencyDomain(omega = 2*np.pi*f, K = K_red, C = C_red, M = M_red)
        eta_hat_store.append(eta_hat)

        # coordinate transformation to obtain the full solution
        resp = V @ eta_hat

        # insert missing nodes with zero, because of the constrains
        resp_all = np.zeros(N*3,dtype=complex)
        resp_all[~Ic] = resp

        #Amplitude displacement
        #P1_resp_z[counter,0] = 20*np.log10(np.abs(resp_all[Iz[N1]]))
        P1_resp_z[i,0] = np.abs(resp_all[Iz[N1]])

        #Phase in degree
        P1_resp_z[i,1] = np.angle(resp_all[Iz[N1]])*180/np.pi

    eta_hat_store = np.asarray(eta_hat_store)

    return(P1_resp_z, eta_hat_store, freq)
```

In [49]:

```
dampingRatio = 0.01 # Damping ratio choosen
```

In [50]:

```
### Rayleigh damping like ex.2
## getting alpha and beta
omegas = np.sqrt(abs(W)) # Collect angular eigenfrq.
omegaCoeffs = np.vstack((0.5/omegas, omegas*0.5)).T # Build coefficent matrix

b = dampingRatio*np.ones(np.shape(omegaCoeffs)[0]) # Right-hand side of omegaCoeffs*alp
haBeta = b

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

dampingRatios = omegaCoeffs @ alphaBeta

start_time = time.time()

## assemble Damping-Matrix for the reduced sys and given aplha and beta for Rayleigh da
mping
alpha = alphaBeta[0]
beta = alphaBeta[1]
Cc = alpha * Mc + beta * Kc

response_ModalCoordinates_frequency = ResponseOverReducedSystem()
P1_resp_z_ray = response_ModalCoordinates_frequency[0]
eta_hat_ray = response_ModalCoordinates_frequency[1]
frequency_ray = response_ModalCoordinates_frequency[2]

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

```
--- 0.043016672134399414 seconds ---
```

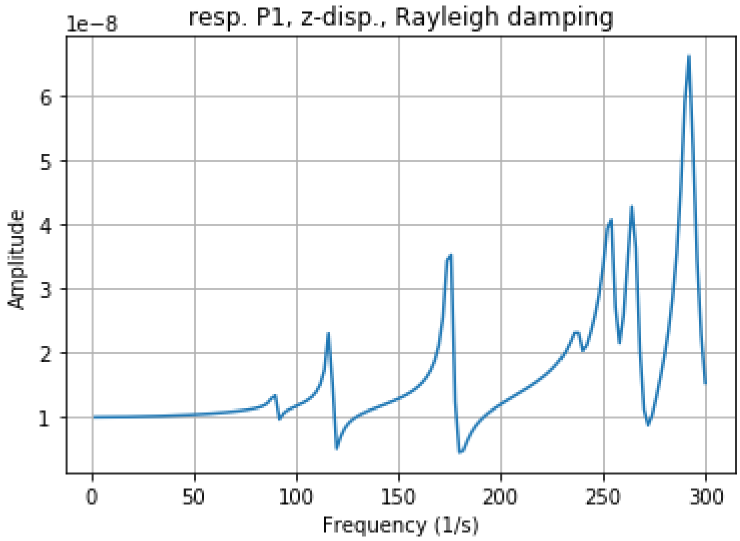
In [51]:

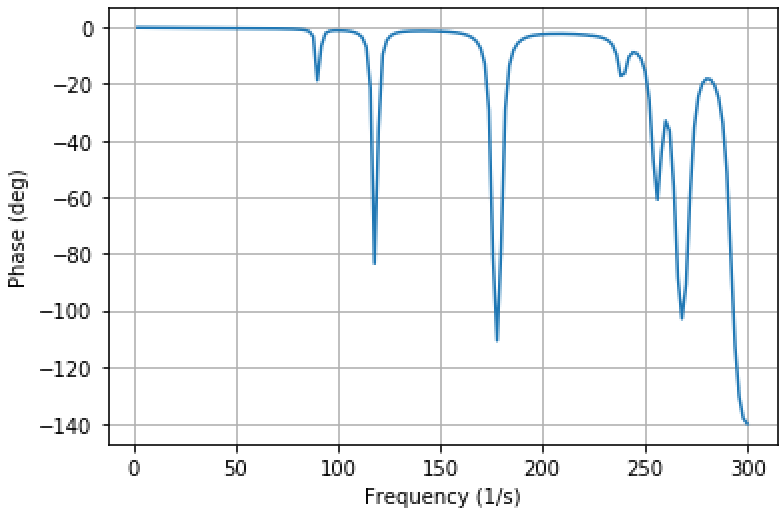
```
### Modal damping
start_time = time.time()

##  assemble Cc-Matrix
container = np.array(2*np.sqrt(W)*dampingRatio)
diagMiddle = np.diag(container)
Cc = V @ diagMiddle @ V.transpose()

response_ModalCoordinates_frequency = ResponseOverReducedSystem()
P1_resp_z_mod = response_ModalCoordinates_frequency[0]
eta_hat_mod = response_ModalCoordinates_frequency[1]
frequency_mod = response_ModalCoordinates_frequency[2]

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

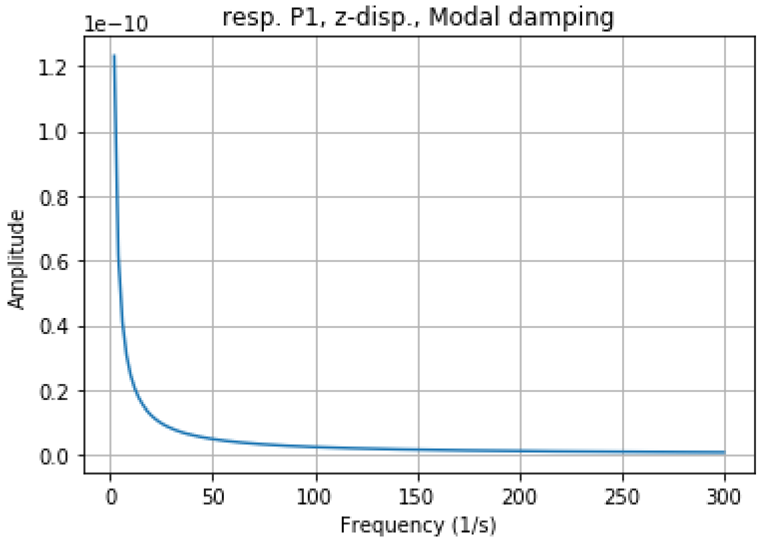
```
--- 0.14645171165466309 seconds ---
```

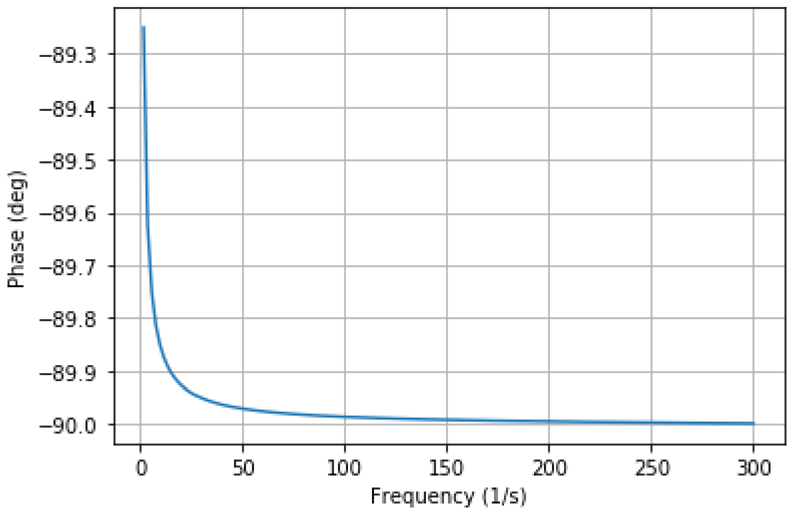
In [52]:

```
### Plot of transfer functions up to 300Hz (Bode-Diag.)

#plot response in z for P1 with Rayleigh damping
plt.plot(frequency_ray, P1_resp_z_ray[:,0])
plt.title('resp. P1, z-disp., Rayleigh damping')
plt.ylabel('Amplitude')
plt.xlabel('Frequency (1/s)')
# plt.xscale('log')
# plt.xlim(1, 1000)
plt.grid(True)
plt.show()

plt.plot(frequency_ray, P1_resp_z_ray[:,1])
plt.ylabel('Phase (deg)')
plt.xlabel('Frequency (1/s)')
# plt.xscale('log')
# plt.xlim(1, 1000)
plt.grid(True)
plt.show()

#plot response in z for P1 with Modal damping
plt.plot(frequency_mod, P1_resp_z_mod[:,0])
plt.title('resp. P1, z-disp., Modal damping')
plt.ylabel('Amplitude')
plt.xlabel('Frequency (1/s)')
# plt.xscale('log')
# plt.xlim(1, 1000)
plt.grid(True)
plt.show()

plt.plot(frequency_mod, P1_resp_z_mod[:,1])
plt.ylabel('Phase (deg)')
plt.xlabel('Frequency (1/s)')
#plt.xscale('log')
#plt.xlim(1, 1000)
plt.grid(True)
plt.show()
```


resp. P1, z-disp., Rayleigh damping
1e−8
Amplitude
Frequency (1/s)
0
50
100
150
200
250
300
1
2
3
4
5
6


Phase (deg)
Frequency (1/s)
0
−20
−40
−60
−80
−100
−120
−140
0
50
100
150
200
250
300





Compare damping models

- what is the difference between modal and Rayleigh damping?
- what happens if you only damp certain modes with modal damping?

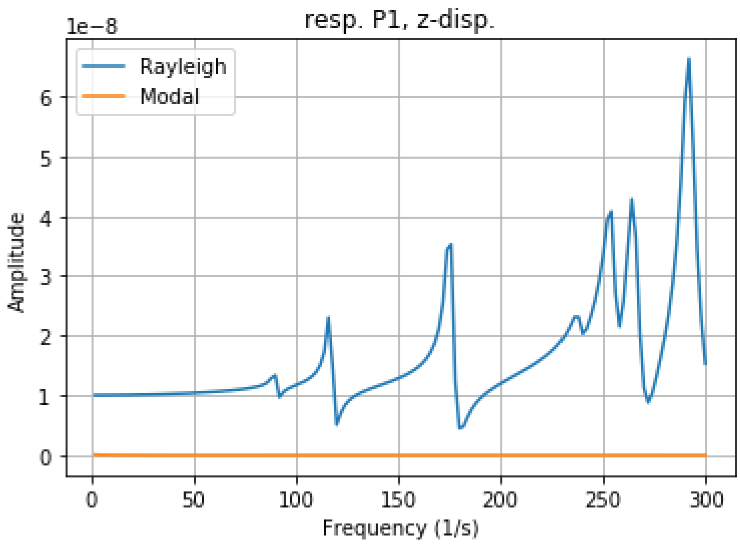
Interpretation

Modal damping acts on all modes the same (if you exert modal damping over all modes), whereas with rayleigh damping only imposes a certain damping-gain at the design points for alpha and beta and the other frequencies get only a portion of the gain respectively more.

If you only damp certain modes with modal damping exactly the damped mode no longer appears.

In [53]:

```
#plot response in z for P1 with Rayleigh damping and Modal damping
plt.plot(frequency_ray, P1_resp_z_ray[:,0], label = 'Rayleigh')
plt.plot(frequency_mod, P1_resp_z_mod[:,0], label = 'Modal')
plt.title('resp. P1, z-disp.')
plt.ylabel('Amplitude')
plt.xlabel('Frequency (1/s)')
# plt.xscale('log')
# plt.xlim(1, 1000)
plt.grid(True)
plt.legend()
plt.show()
```

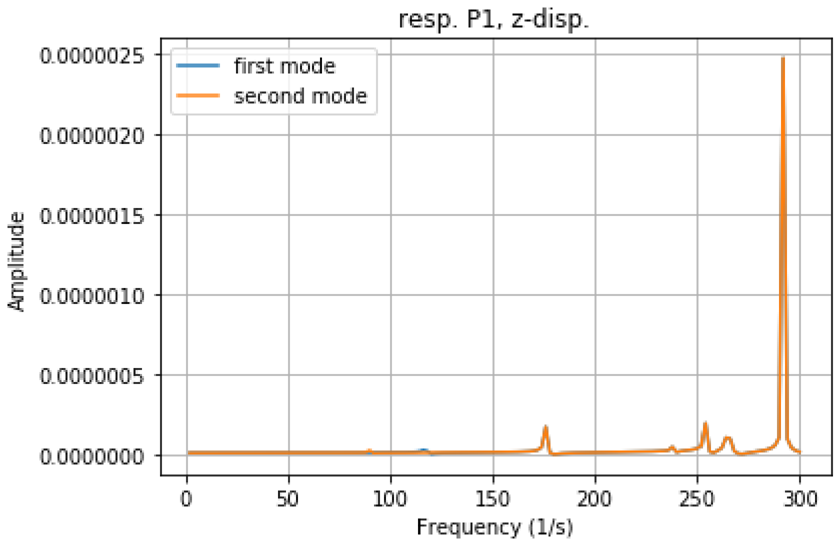


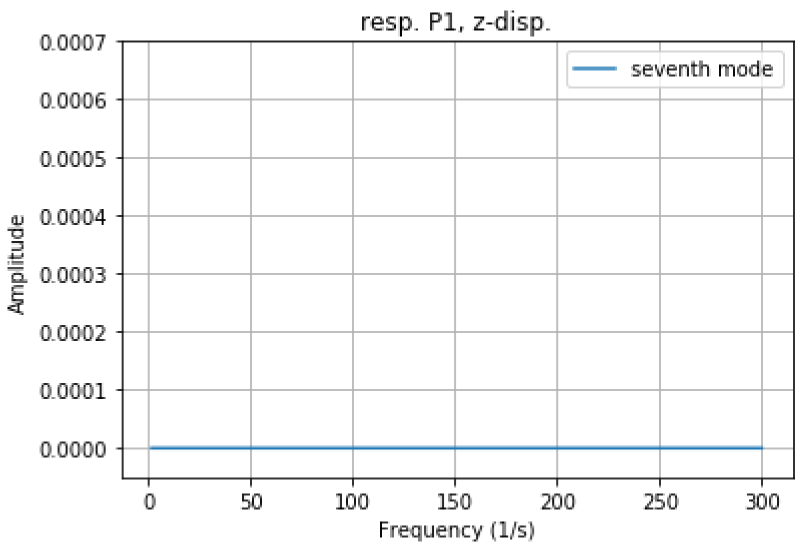
In [54]:

```
## only damp certain modes with modal damping

### assemble Cc-Matrix; only damp first mode with modal damping
container = np.zeros(len(W))
which_mode = 0
container[which_mode] = 2*np.sqrt(W[which_mode])*dampingRatio
diagMiddle = np.diag(container)
Cc = V @ diagMiddle @ V.transpose()

response_ModalCoordinates_frequency = ResponseOverReducedSystem()
P1_resp_z_mod_first = response_ModalCoordinates_frequency[0]
eta_hat_mod_first = response_ModalCoordinates_frequency[1]
frequency_first = response_ModalCoordinates_frequency[2]

### assemble Cc-Matrix; only damp second mode with modal damping
container = np.zeros(len(W))
which_mode = 1
container[which_mode] = 2*np.sqrt(W[which_mode])*dampingRatio
diagMiddle = np.diag(container)
Cc = V @ diagMiddle @ V.transpose()

response_ModalCoordinates_frequency = ResponseOverReducedSystem()
P1_resp_z_mod_second = response_ModalCoordinates_frequency[0]
eta_hat_mod_second = response_ModalCoordinates_frequency[1]
frequency_second = response_ModalCoordinates_frequency[2]

### assemble Cc-Matrix; only damp the seventh mode with modal damping
container = np.zeros(len(W))
which_mode = 6
container[which_mode] = 2*np.sqrt(W[which_mode])*dampingRatio
diagMiddle = np.diag(container)
Cc = V @ diagMiddle @ V.transpose()

response_ModalCoordinates_frequency = ResponseOverReducedSystem()
P1_resp_z_mod_seventh = response_ModalCoordinates_frequency[0]
eta_hat_mod_seventh = response_ModalCoordinates_frequency[1]
frequency_seventh = response_ModalCoordinates_frequency[2]
```

In [55]:

```
#plot response in z for P1 with Modal damping on certain modes
plt.plot(frequency_ray, P1_resp_z_mod_first[:,0], label = 'first mode')
plt.plot(frequency_mod, P1_resp_z_mod_second[:,0], label = 'second mode')
# plt.plot(frequency_mod, P1_resp_z_mod_seventh[:,0], label = 'seventh mode')
plt.title('resp. P1, z-disp.')
plt.ylabel('Amplitude')
plt.xlabel('Frequency (1/s)')
# plt.xscale('log')
# plt.xlim(1, 1000)
plt.grid(True)
plt.legend()
plt.show()

#plot response in z for P1 with Modal damping on seventh mode (last under 300Hz)
plt.plot(frequency_mod, P1_resp_z_mod_seventh[:,0], label = 'seventh mode')
plt.title('resp. P1, z-disp.')
plt.ylabel('Amplitude')
plt.xlabel('Frequency (1/s)')
# plt.xscale('log')
plt.ylim(-0.00005, 0.0007)
plt.grid(True)
plt.legend()
plt.show()
```





In [56]:

```
# Output of the natural frequencies for illustration
print(np.sqrt(W)/2/np.pi)
```

```
[ 90.27687698 117.31775427 175.69534125 238.50182961 254.41077978
 265.08306408 292.08229899 359.70245276 383.69546858 460.57030902]
```

Modal contribution

- compute the modal contribution factors for each mode and plot them over the frequency
- When is which mode important?

Interpretation

Whenever the frequency is close to the natural frequency of one mode, this mode is the most important one.

In [57]:

```
### Modal Contribution factor for each mode
frequency = frequency_ray
eta = eta_hat_ray

rho = np.zeros([k, len(frequency)],dtype=float)

for i,f in enumerate(frequency):
    data = np.abs(eta[i,:])

    rho[:,i] = (data)/np.linalg.norm(data)
    rho[:,i] = rho[:,i]/np.sum(rho[:,i])
```

In [58]:

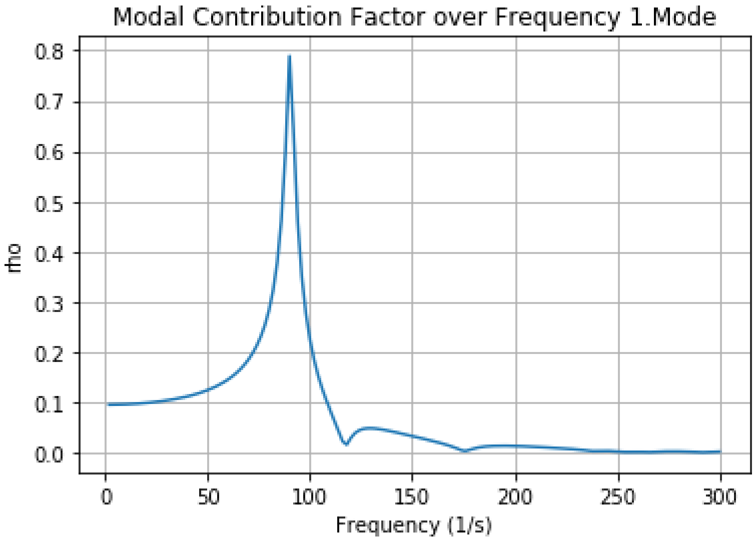
```
np.sum(np.abs(rho[:,1]))
```

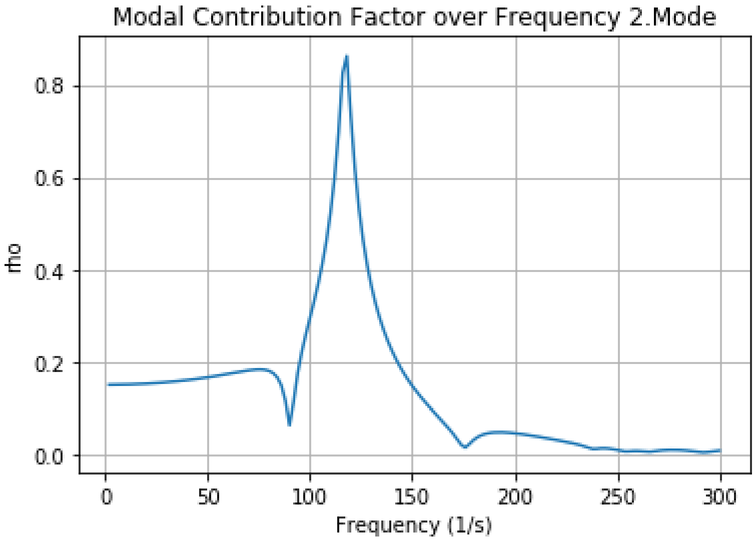
Out[58]:

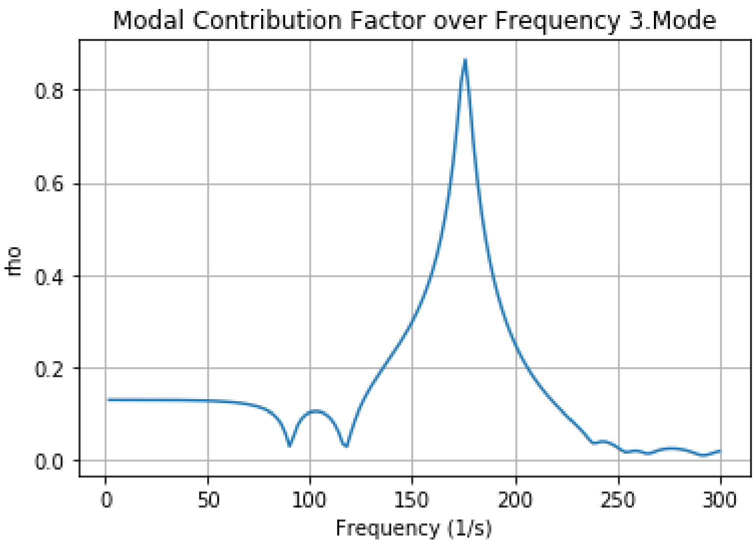
```
1.0000000000000002
```

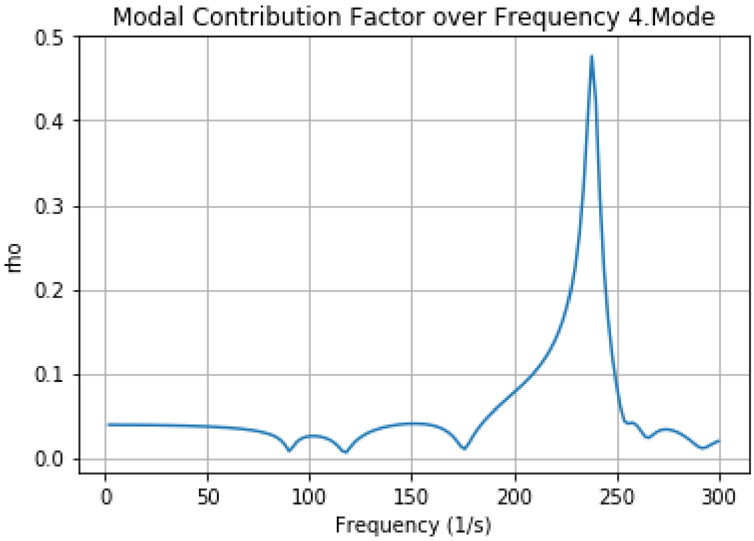
In [59]:

```
#plot modal Contribution Factor over the frequency 1.Mode

for i,values in enumerate(rho):
    plt.plot(frequency, rho[i,:])
    plt.title('Modal Contribution Factor over Frequency ' + str(i+1) + '.Mode')
    plt.ylabel('rho')
    plt.xlabel('Frequency (1/s)')
    plt.grid(True)
    plt.show()
```

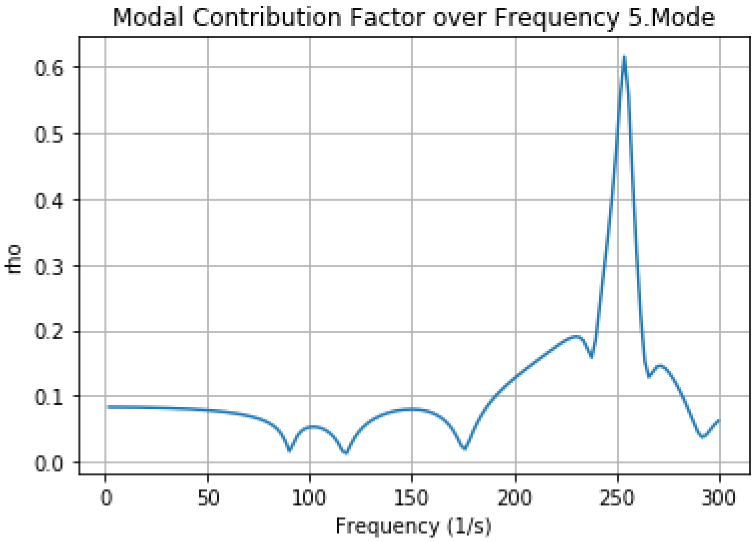

Modal Contribution Factor over Frequency 1.Mode
rho
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
50
100
150
200
250
300
Frequency (1/s)


Modal Contribution Factor over Frequency 2.Mode
rho
0.0
0.2
0.4
0.6
0.8
0
50
100
150
200
250
300
Frequency (1/s)

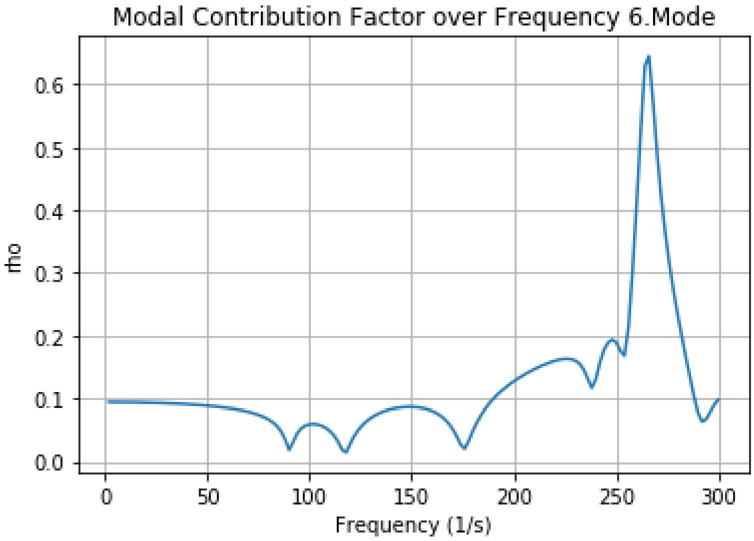

Modal Contribution Factor over Frequency 3.Mode
rho
0.0
0.2
0.4
0.6
0.8
0
50
100
150
200
250
300
Frequency (1/s)



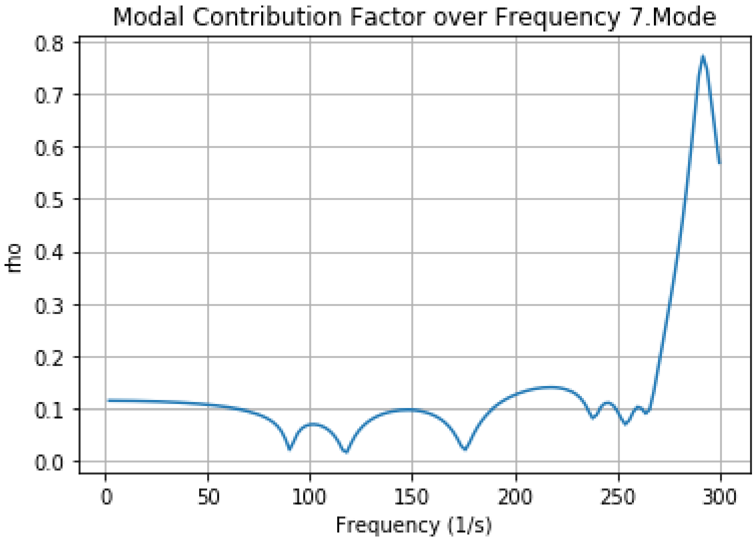
Modal Contribution Factor over Frequency 4.Mode
rho
Frequency (1/s)
0.0
0.1
0.2
0.3
0.4
0.5
0
50
100
150
200
250
300

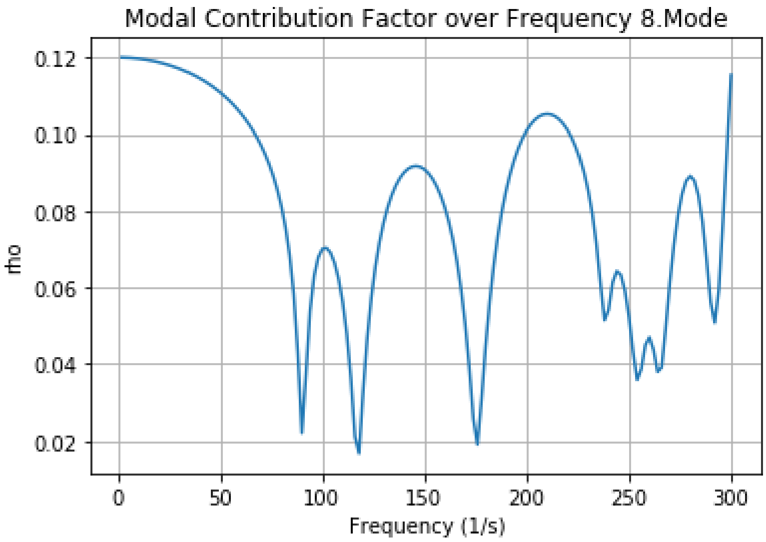


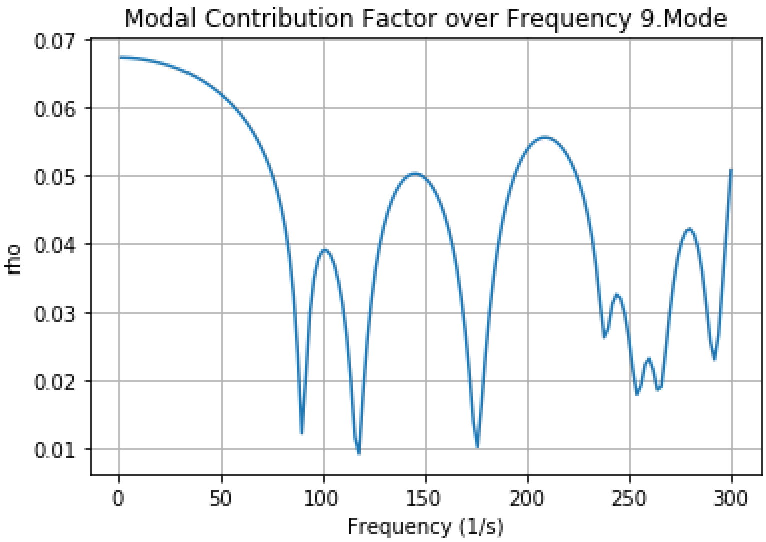
Modal Contribution Factor over Frequency 5.Mode
rho
Frequency (1/s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
50
100
150
200
250
300

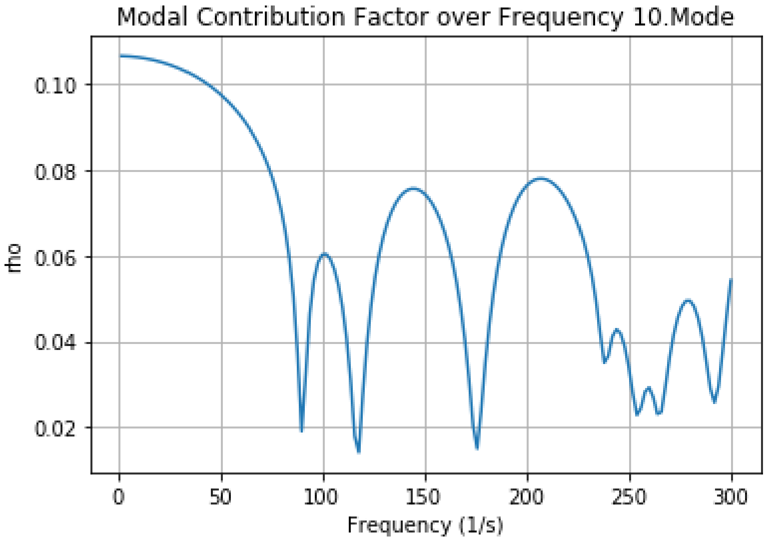


Modal Contribution Factor over Frequency 6.Mode
rho
Frequency (1/s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
50
100
150
200
250
300


Modal Contribution Factor over Frequency 7.Mode
rho
Frequency (1/s)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
50
100
150
200
250
300


Modal Contribution Factor over Frequency 8.Mode
rho
Frequency (1/s)
0.02
0.04
0.06
0.08
0.10
0.12
0
50
100
150
200
250
300


Modal Contribution Factor over Frequency 9.Mode
rho
Frequency (1/s)
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0
50
100
150
200
250
300


Modal Contribution Factor over Frequency 10.Mode
rho
0.10
0.08
0.06
0.04
0.02
0
50
100
150
200
250
300
Frequency (1/s)

In [60]:

```
## pLot modal contribution of the modes
myRange = np.arange(0,len(frequency),5)

frq_label = frequency[myRange]
frq_label = frq_label.astype(str)
frq_label = np.char.add('freq ',frq_label)

fig, ax = plt.subplots(figsize=(15,10))
for i,f in enumerate(myRange):
    ax.plot(rho[:,f],label=frq_label[i])

ax.set_xlabel('modes')
ax.set_ylabel('modal contribution')
plt.title('modal contribution of all modes')
plt.grid(True)
plt.legend(loc='right')
plt.show()
```