In [76]:
```
import numpy as np
from scipy.io import mminfo,mmread
import matplotlib.pyplot as plt
%matplotlib inline

from scipy.linalg import eig
from numpy import sqrt, dot, sum, abs, diag, array, pi
from scipy import sparse

from numpy.fft import rfft, rfftfreq

import sys
np.set_printoptions(threshold=sys.maxsize)
# np.set_printoptions(threshold=20)
import warnings
warnings.filterwarnings('ignore')
```

Exercise 5 - Identification Techniques

This exercise is focused on identifying modal parameters from experiments.

Measurement data from a numerical impact hammer experiment on the plate supported by flexible springs is provided in the file `measurement.txt` . Time histories of acceleration at 24 points and of the force at the impact point were measured. The point-coordinates are given in `measurement-coordinates.txt` .

You can load the data with

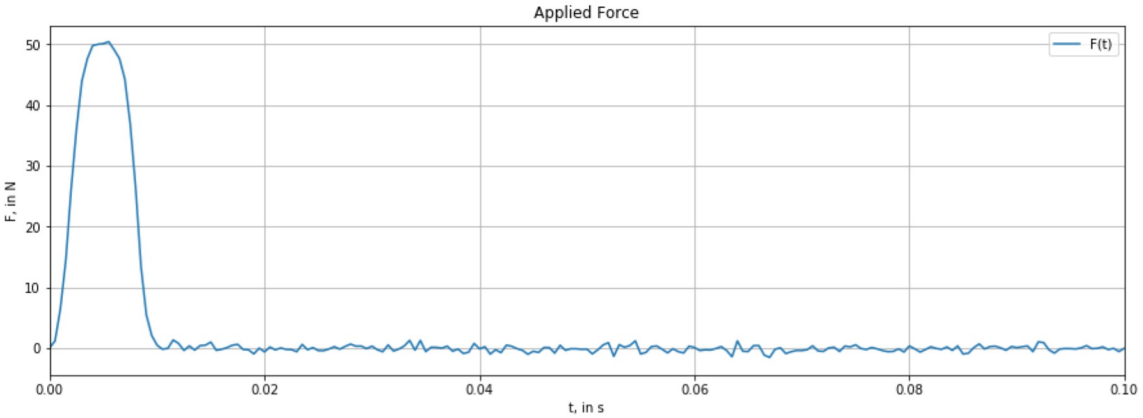
```
from numpy import loadtxt
data = loadtxt('measurement.txt')
```

In [3]:
```
from numpy import loadtxt
data = loadtxt('measurement.txt')
```

- Plot the time history for the force as well as its frequency spectrum.

In [4]:
```
t = data[:,0] #Time vector
F = data[:,1] #Force vector
```

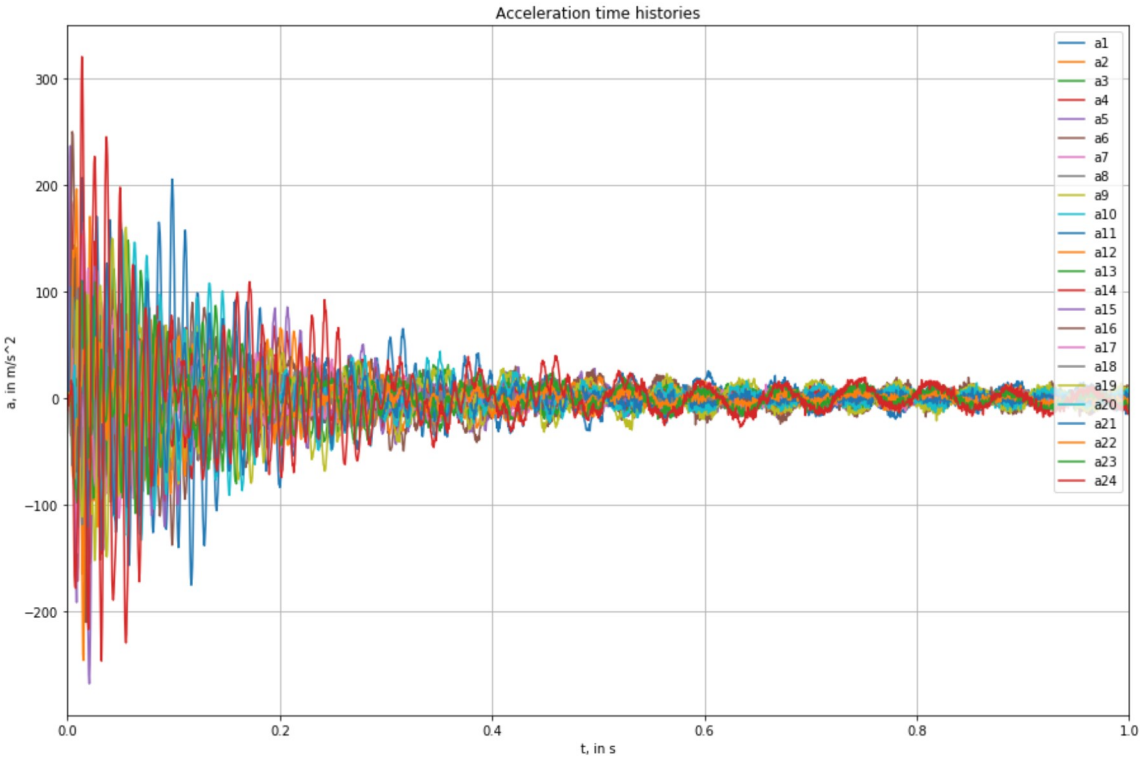
In [5]:
```
plt.figure(figsize = [15,5])
plt.plot(t,F,label = 'F(t)')
plt.xlabel('t, in s')
plt.ylabel('F, in N')
plt.title('Applied Force')
plt.xlim([0,0.1])
plt.legend()
plt.grid()
```



In [6]:
```
frequencies = rfftfreq(len(t),t[1] - t[0])
F_rfft = rfft(F)
```

- Plot the measured acceleration time histories and frequency spectra for the 4 corner points and the drive point (where the force is applied). Make sure to select appropriate time and frequency limits to only show useful data.

In [7]:
```
plt.figure(figsize = [15,10])
plt.xlabel('t, in s')
plt.ylabel('a, in m/s^2')
plt.title('Acceleration time histories')
plt.xlim([0,1])

for i in range(2,26) :
    plt.plot(t,data[:,i],label = 'a' + str(i-1))

plt.legend(loc = 1)
plt.grid()
```



Now we calculate the frequeny spectra of the 4 corner points. Form the Numerical Experiment Jupyter Notebook we acquire following information:

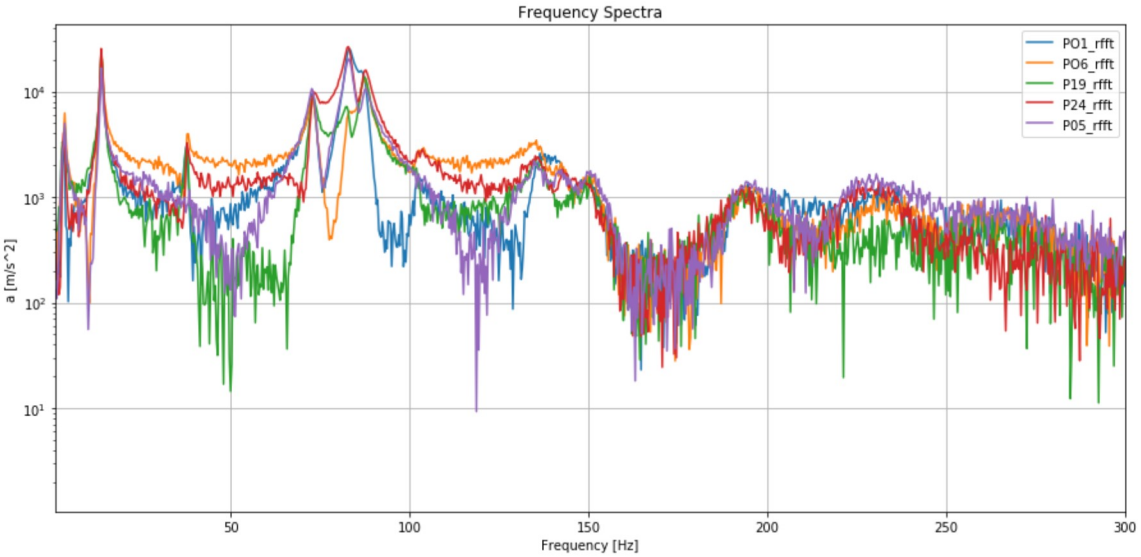
- **P01** - Left lower corner
- **P06** - Right lower corner
- **P19** - Left top corner
- **P24** - Right top corner

For the injection point of the force: **P05**

In [8]:
```
PO1_rfft = rfft(data[:,2])
PO6_rfft = rfft(data[:,7])
P19_rfft = rfft(data[:,20])
P24_rfft = rfft(data[:,25])

P05_rfft = rfft(data[:,6])
```

Now let's plot the frequency spectra

In [9]:
```
plt.figure(figsize = [15,7])

plt.semilogy(frequencies,np.abs(PO1_rfft), label='PO1_rfft')
plt.semilogy(frequencies,np.abs(PO6_rfft), label='PO6_rfft')
plt.semilogy(frequencies,np.abs(P19_rfft), label='P19_rfft')
plt.semilogy(frequencies,np.abs(P24_rfft), label='P24_rfft')
plt.semilogy(frequencies,np.abs(P05_rfft), label='P05_rfft')

plt.ylabel('a [m/s^2]')
plt.xlabel('Frequency [Hz]')
plt.xlim(1,300)
plt.title('Frequency Spectra')
plt.legend()
plt.grid()
```



- Compute the accelerance and plot it for the drive point.

Make sure to apply a suitable window to the data, to counteract the effect of poor signal to noise ratio in parts of the singal. For decaying signal a decaring exponential window is suitable, e.g. use

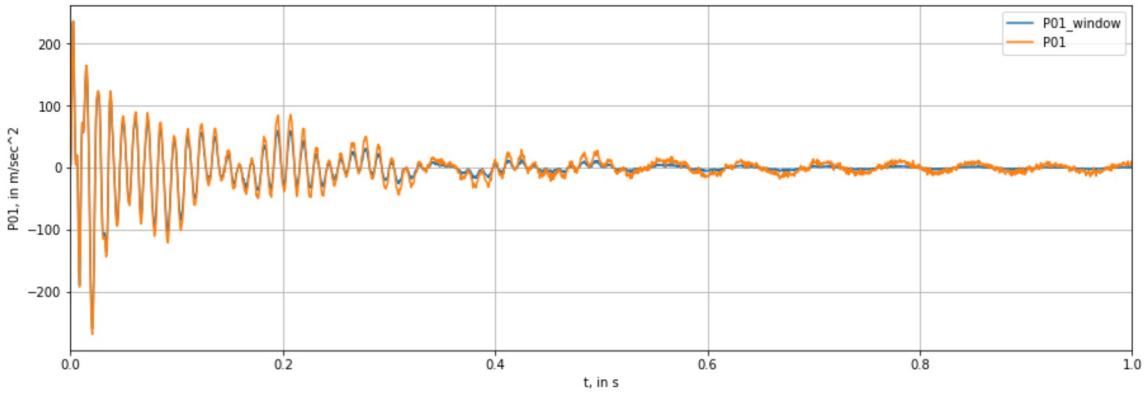
```
from scipy.signal import windows
frac = 1e-3 # choose something useful
W = windows.exponential(len(t), 0, -(len(t)-1) / np.log(frac), False)
```

Multiply the window to the data, before you do the FFT.

In [10]:
```
from scipy.signal import windows
frac = 5e-3 # choose something useful
W = windows.exponential(len(t), 0, -(len(t)-1) / np.log(frac), False)
```

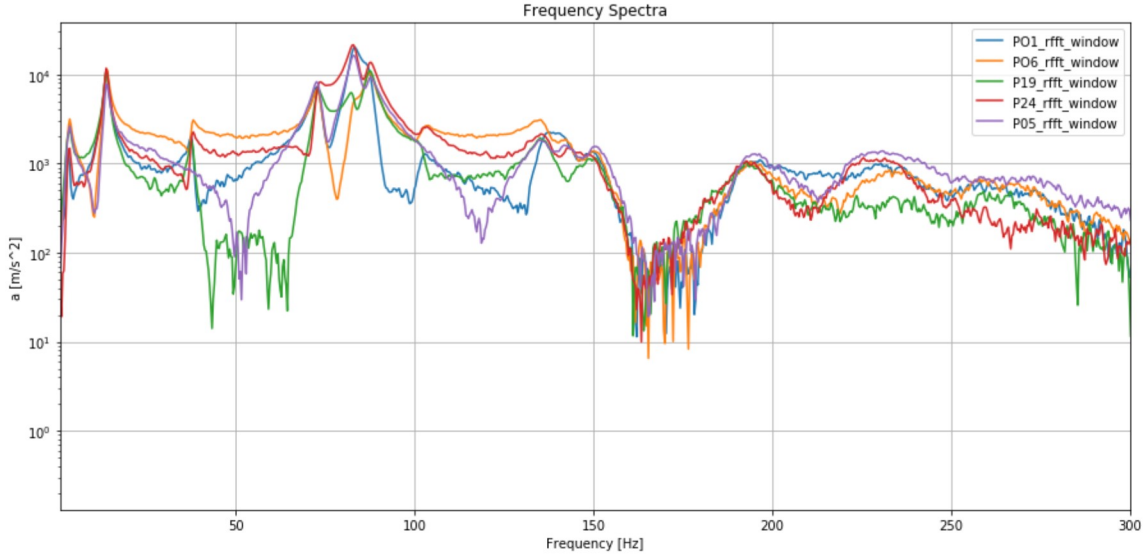
Let's first compare for one point how the window function influences the acceleration

In [11]:
```
plt.figure(figsize = [15,5])
plt.plot(t,data[:,6]*W, label = "P01_window")
plt.plot(t,data[:,6], label = "P01")
plt.xlabel('t, in s')
plt.ylabel('P01, in m/sec^2')
plt.xlim([0,1])
plt.legend()
plt.grid()
```



In [12]:
```
PO1_rfft_window = rfft(data[:,2]*W)
PO6_rfft_window = rfft(data[:,7]*W)
P19_rfft_window = rfft(data[:,20]*W)
P24_rfft_window = rfft(data[:,25]*W)

P05_rfft_window = rfft(data[:,6]*W)
```

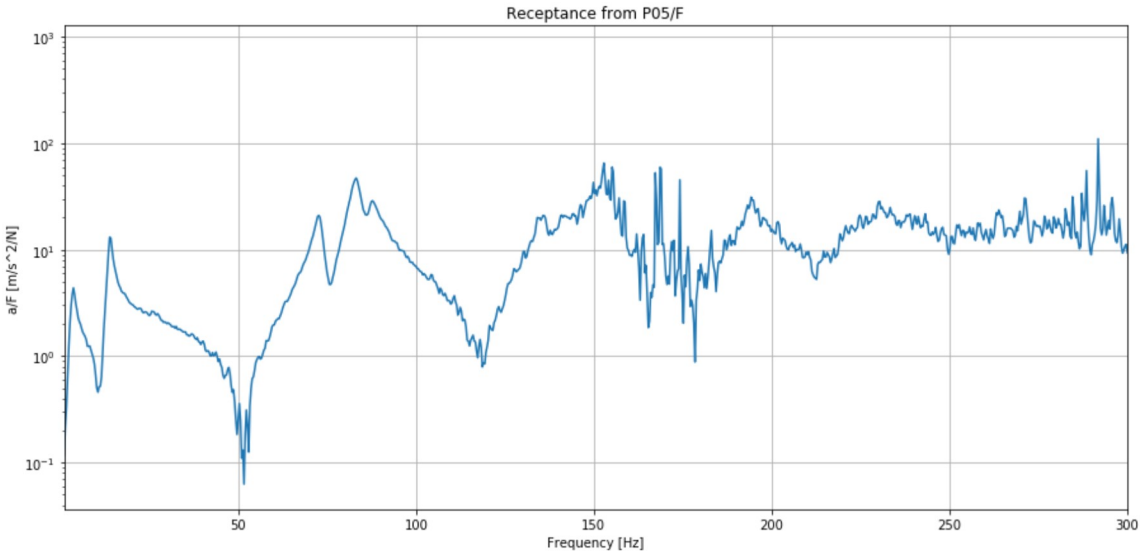
In [13]:
```
plt.figure(figsize = [15,7])

plt.semilogy(frequencies,np.abs(PO1_rfft_window), label='PO1_rfft_window')
plt.semilogy(frequencies,np.abs(PO6_rfft_window), label='PO6_rfft_window')
plt.semilogy(frequencies,np.abs(P19_rfft_window), label='P19_rfft_window')
plt.semilogy(frequencies,np.abs(P24_rfft_window), label='P24_rfft_window')
plt.semilogy(frequencies,np.abs(P05_rfft_window), label='P05_rfft_window')

plt.ylabel('a [m/s^2]')
plt.xlabel('Frequency [Hz]')
plt.xlim(1,300)
plt.title('Frequency Spectra')
plt.legend()
plt.grid()
```



- Compute the receptance from the accelerance and plot it for the drive point.

In [14]:
```
P05_receptance = rfft(data[:,6]*W)/rfft(data[:,1]*W)
```

In [15]:
```
plt.figure(figsize = [15,7])

plt.semilogy(frequencies,np.abs(P05_receptance))

plt.ylabel('a/F [m/s^2/N]')
plt.xlabel('Frequency [Hz]')
plt.xlim(1,300)
plt.title('Receptance from P05/F')
plt.grid()
```



Task1: Find Peaks in the Transfer Functions

Use the Maximum Amplitude, Maximum Quadrature or Maximum Quadrature Component method to find all natural frequencies up to 150Hz.

For low frequency peaks it might me best to look at receptance curves, wheras for high frequenies the peaks in the accelerance might be more pronounced. Some automatic peak finding algorithms are implemented in scipy. A useful and rather robust automatic peak-detection tool is the continuous wavelet transform, e.g. use

```
from scipy.signal import find_peaks, find_peaks_cwt
```

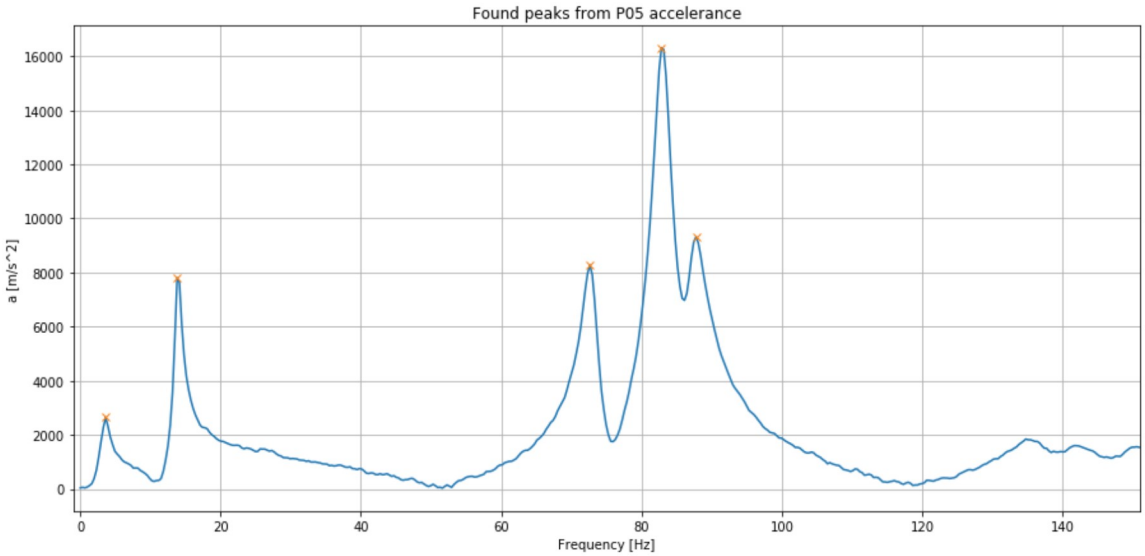
Often the automatic selection has to be fine tuned by hand.

- Use an automatic tool to detect peaks in the drive-point accelerance, and plot the found peaks.
- Then adapt the result by hand (add shift peaks) until you have found all natural frequencies.
- Plot the magnitude, real and imaginary part of the fransfer functions around each peak in a suitable frequency range (~10Hz) to verify the peak location: all transfer functions in one plot, one plot per part, one plot per peak

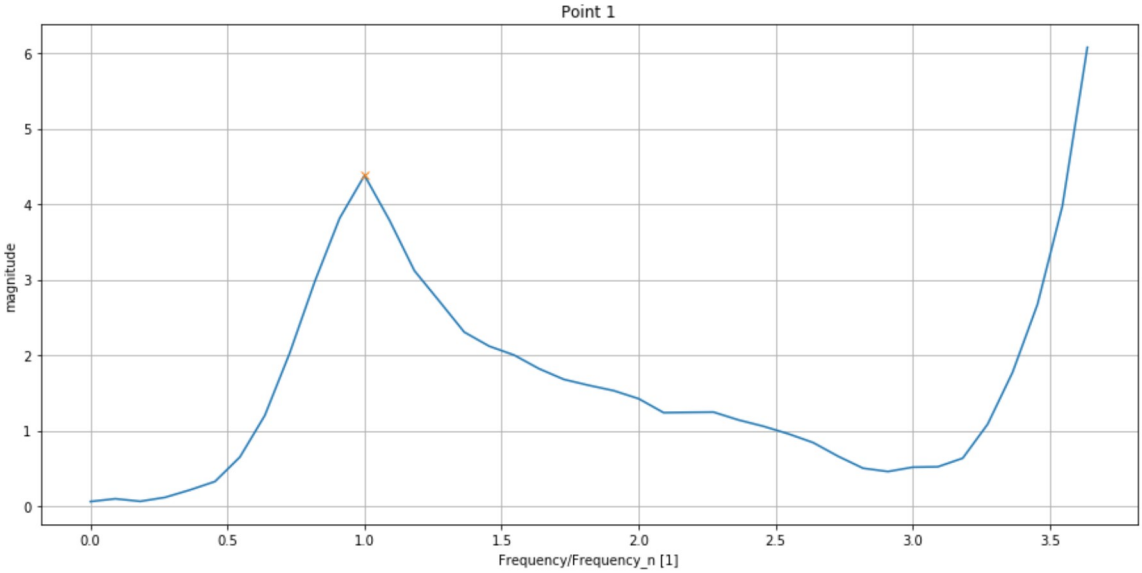
In [16]:
```
from scipy.signal import find_peaks, find_peaks_cwt
```

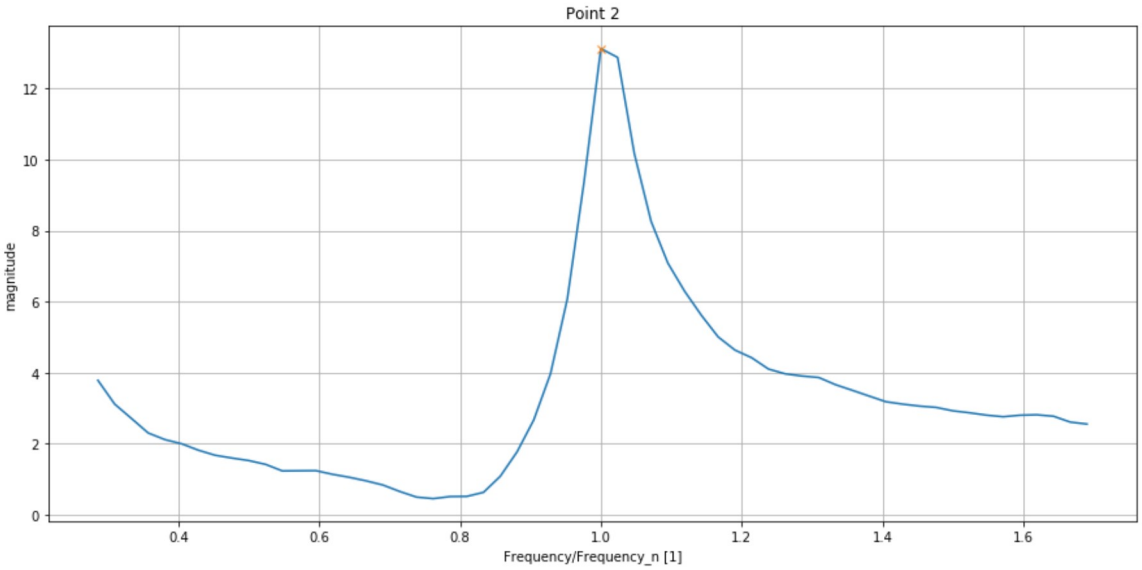
In [17]:
```
# Use an automatic tool to detect peaks in the drive-point accelerance
peaks_idx_P05_rfft_window, _ = find_peaks(np.abs(P05_rfft_window), height=2000)
# The height is set so that only realistic peaks are taken into account
```

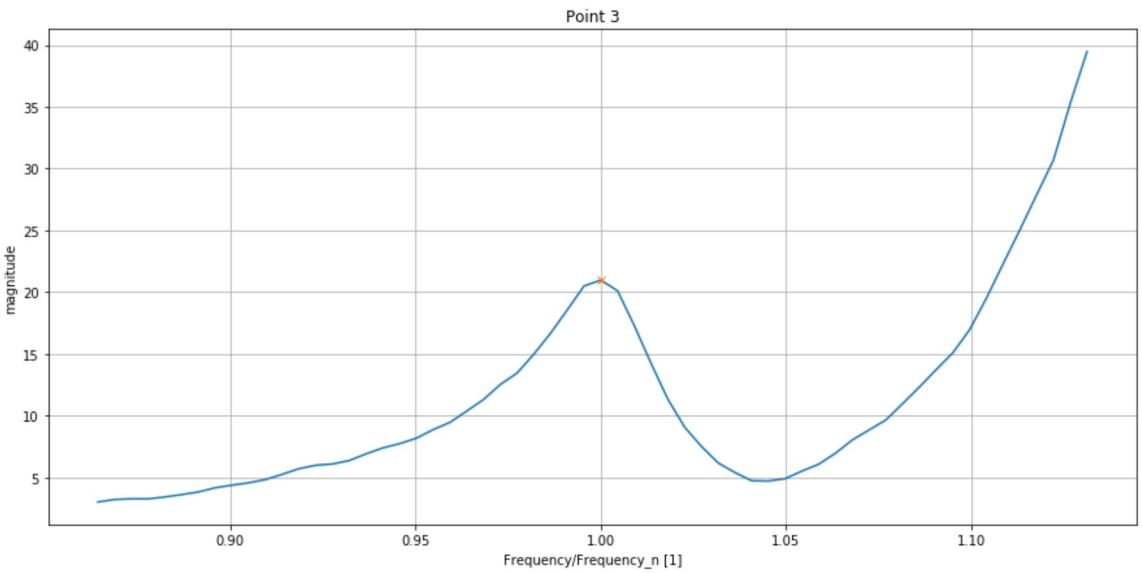
In [18]:
```
plt.figure(figsize = [15,7])
plt.ylabel('a [m/s^2]')
plt.xlabel('Frequency [Hz]')
plt.xlim(-1,151)
plt.title('Found peaks from P05 accelerance')
plt.grid()
plt.plot(frequencies, np.abs(P05_rfft_window))
plt.plot(frequencies[peaks_idx_P05_rfft_window], np.abs(P05_rfft_window)[peaks_i
dx_P05_rfft_window], 'x')
plt.show()
```

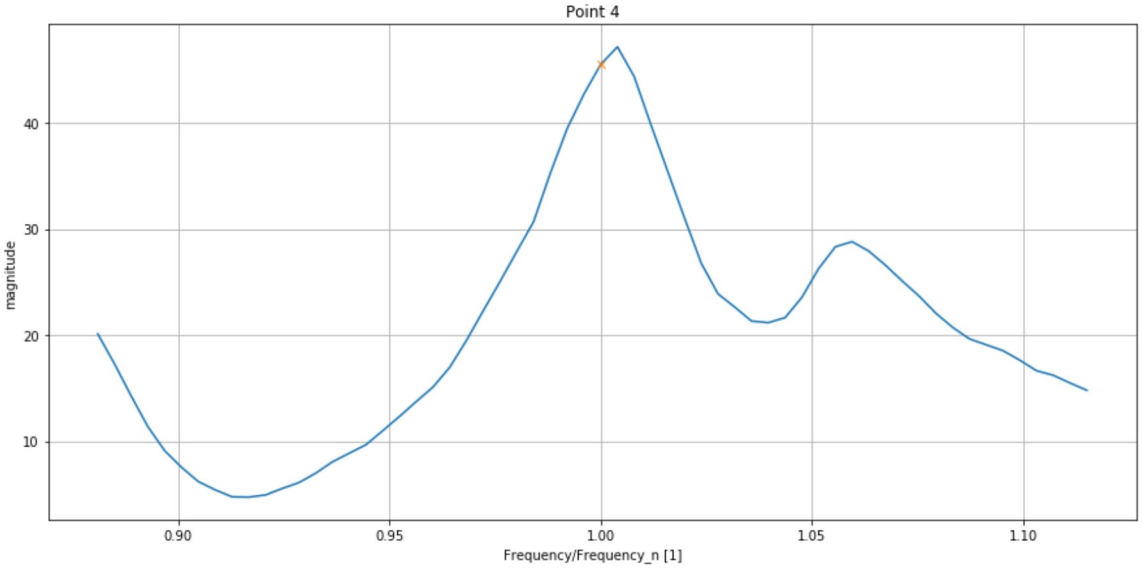


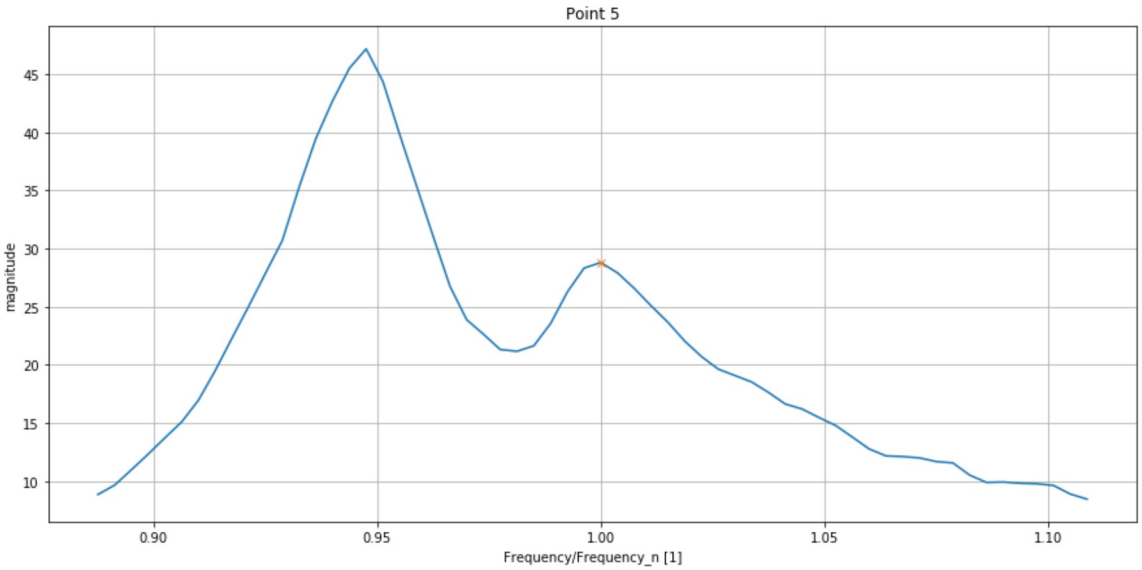
In [19]:
```
# Plot the magnitude
bound = 10;

for n,i in enumerate(peaks_idx_P05_rfft_window):
    plt.figure(figsize = [15,7])

    lower_freq = frequencies[i]-bound
    upper_freq = frequencies[i]+bound

    lower_idx = (np.abs(frequencies - lower_freq)).argmin()
    upper_idx = (np.abs(frequencies - upper_freq)).argmin()

    plt.plot(frequencies[lower_idx:upper_idx]/frequencies[i], np.abs(P05_recepta
nce[lower_idx:upper_idx]))
    plt.plot(frequencies[i]/frequencies[i], np.abs(P05_receptance[i]), 'x')

    plt.ylabel('magnitude')
    plt.xlabel('Frequency/Frequency_n [1]')
    plt.title('Point '+ str(n+1))
    plt.grid()
```


Point 1
magnitude
Frequency/Frequency_n [1]

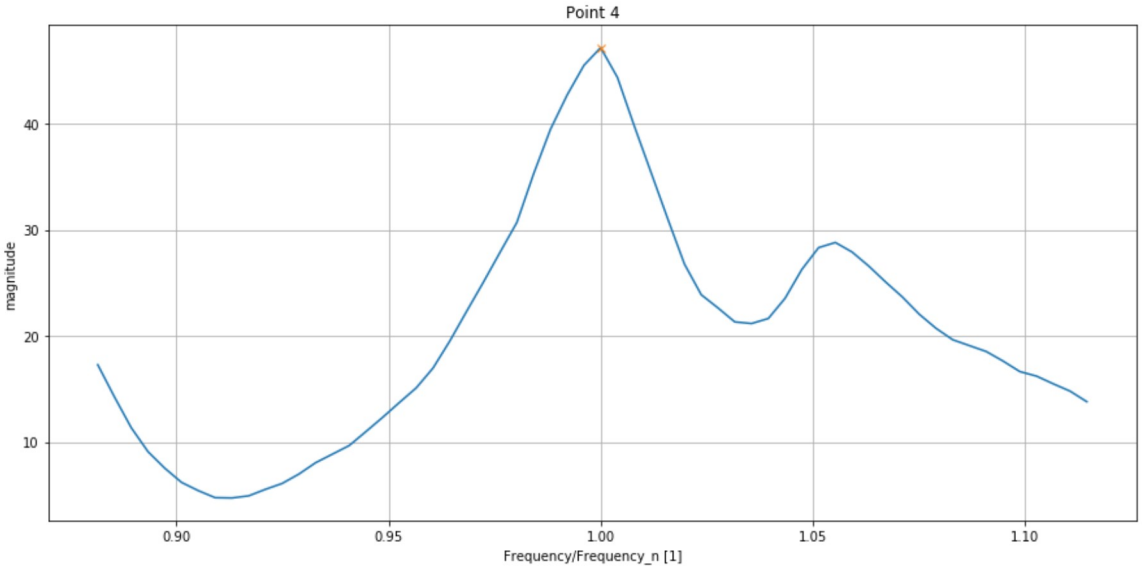

Point 2
magnitude
Frequency/Frequency_n [1]


Point 3
magnitude
Frequency/Frequency_n [1]

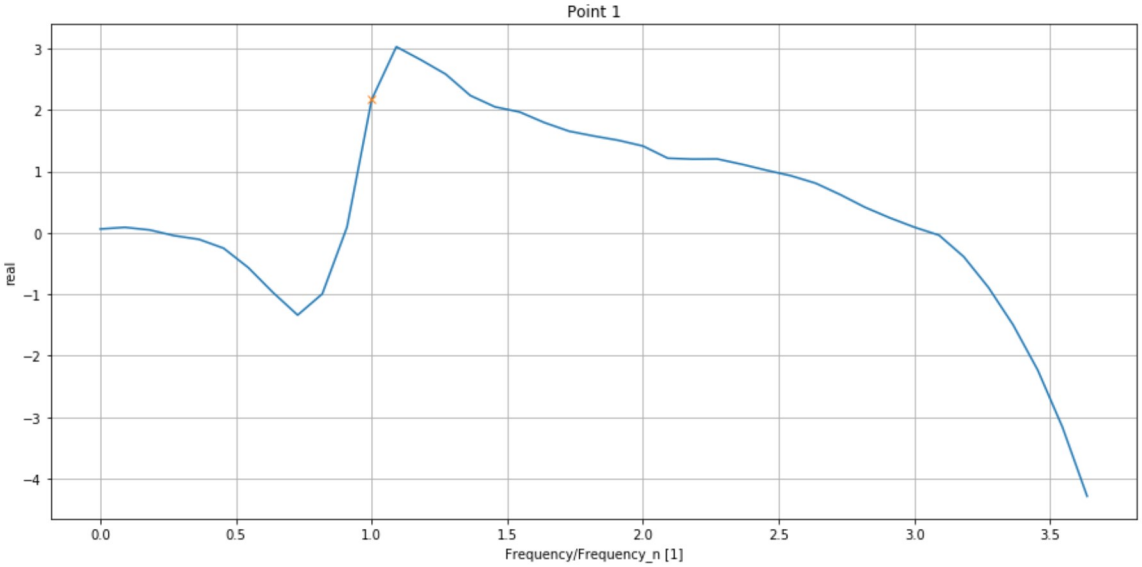

Point 4
magnitude
Frequency/Frequency_n [1]
40
30
20
10
0.90
0.95
1.00
1.05
1.10

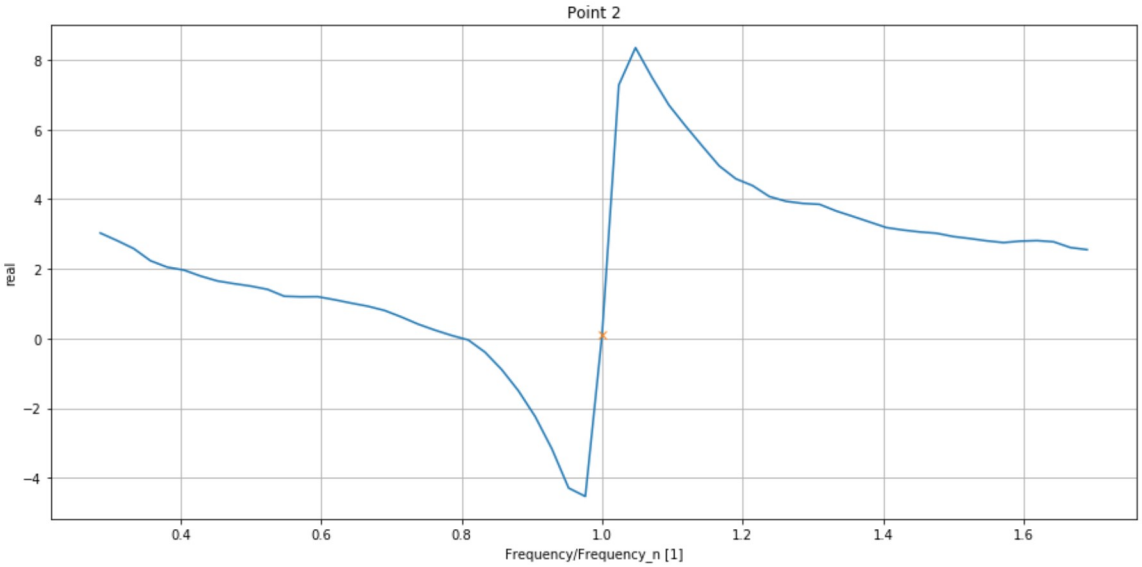

Point 5
magnitude
Frequency/Frequency_n [1]
45
40
35
30
25
20
15
10
0.90
0.95
1.00
1.05
1.10

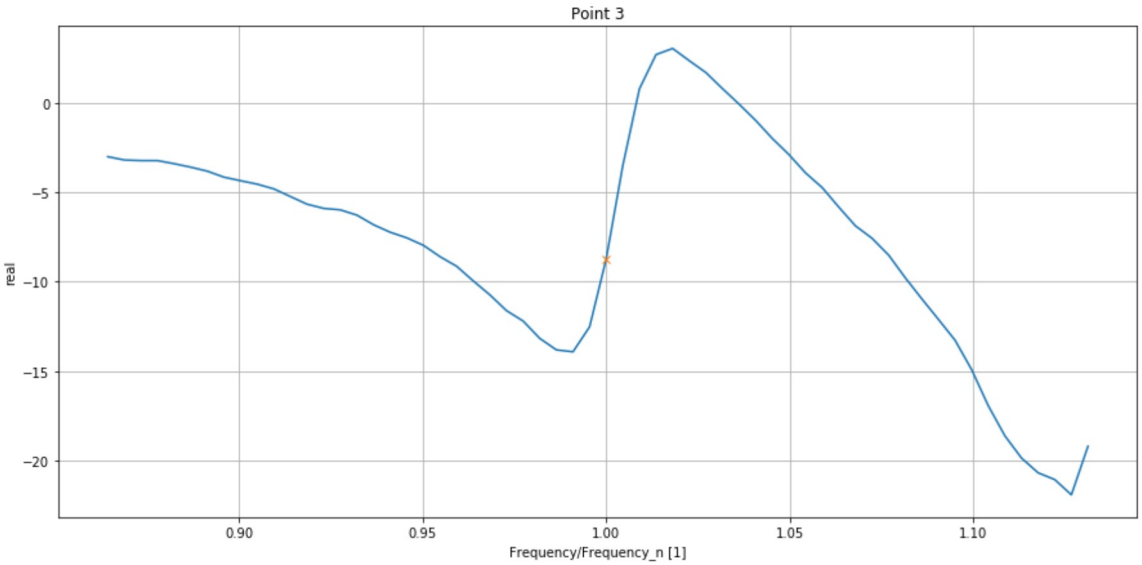
In [20]:
```
# shift point 4 to the peak (Peak-Amplitude Methode)
peaks_idx_P05_rfft_window[3] = 253


plt.figure(figsize = [15,7])

lower_freq = frequencies[peaks_idx_P05_rfft_window[3]]-bound
upper_freq = frequencies[peaks_idx_P05_rfft_window[3]]+bound

lower_idx = (np.abs(frequencies - lower_freq)).argmin()
upper_idx = (np.abs(frequencies - upper_freq)).argmin()

plt.plot(frequencies[lower_idx:upper_idx]/frequencies[peaks_idx_P05_rfft_window
[3]], np.abs(P05_receptance[lower_idx:upper_idx]))
plt.plot(frequencies[peaks_idx_P05_rfft_window[3]]/frequencies[peaks_idx_P05_rff
t_window[3]], np.abs(P05_receptance[peaks_idx_P05_rfft_window[3]]), 'x')

plt.ylabel('magnitude')
plt.xlabel('Frequency/Frequency_n [1]')
plt.title('Point '+ str(4))
plt.grid()
```

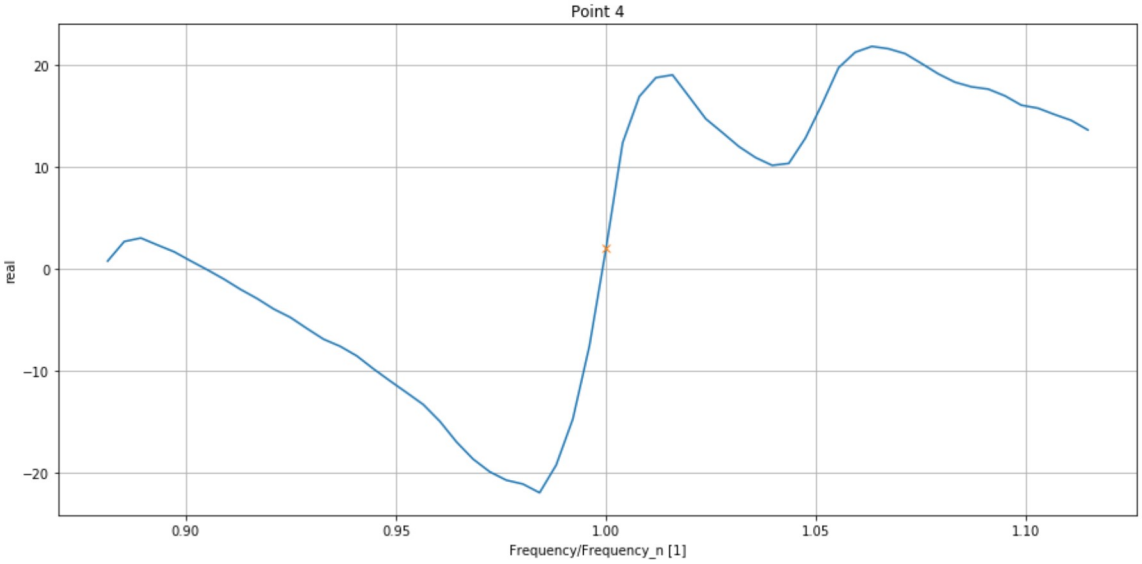


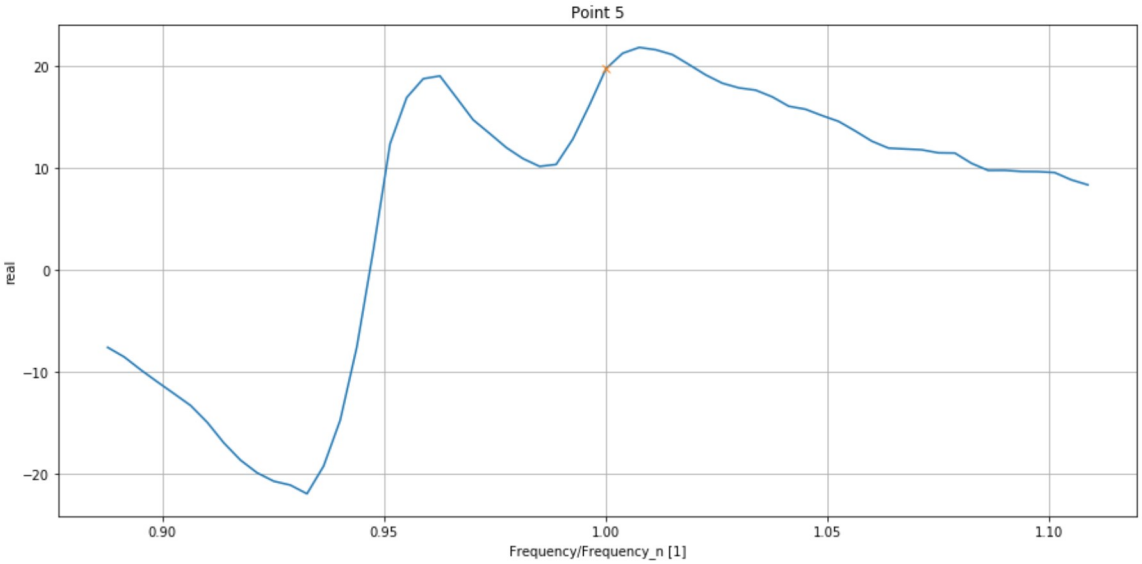
In [21]:
```
# Plot the real part
for n,i in enumerate(peaks_idx_P05_rfft_window):
    plt.figure(figsize = [15,7])

    lower_freq = frequencies[i]-bound
    upper_freq = frequencies[i]+bound

    lower_idx = (np.abs(frequencies - lower_freq)).argmin()
    upper_idx = (np.abs(frequencies - upper_freq)).argmin()

    plt.plot(frequencies[lower_idx:upper_idx]/frequencies[i], np.real(P05_recept
ance[lower_idx:upper_idx]))
    plt.plot(frequencies[i]/frequencies[i], np.real(P05_receptance[i]), 'x')

    plt.ylabel('real')
    plt.xlabel('Frequency/Frequency_n [1]')
    plt.title('Point '+ str(n+1))
    plt.grid()
```


Point 1
real
Frequency/Frequency_n [1]

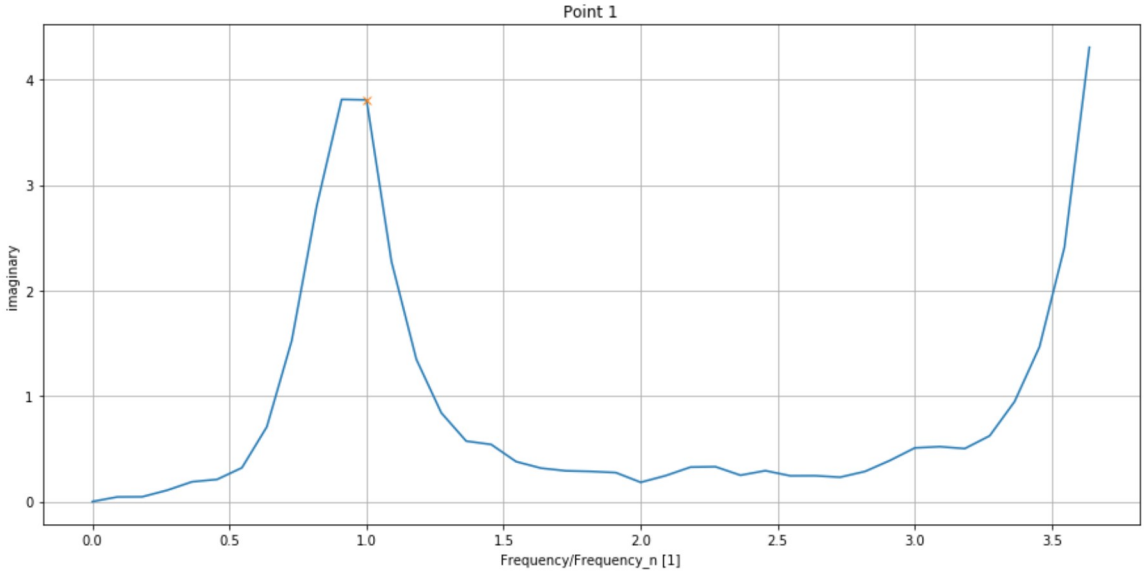

Point 2
real
Frequency/Frequency_n [1]

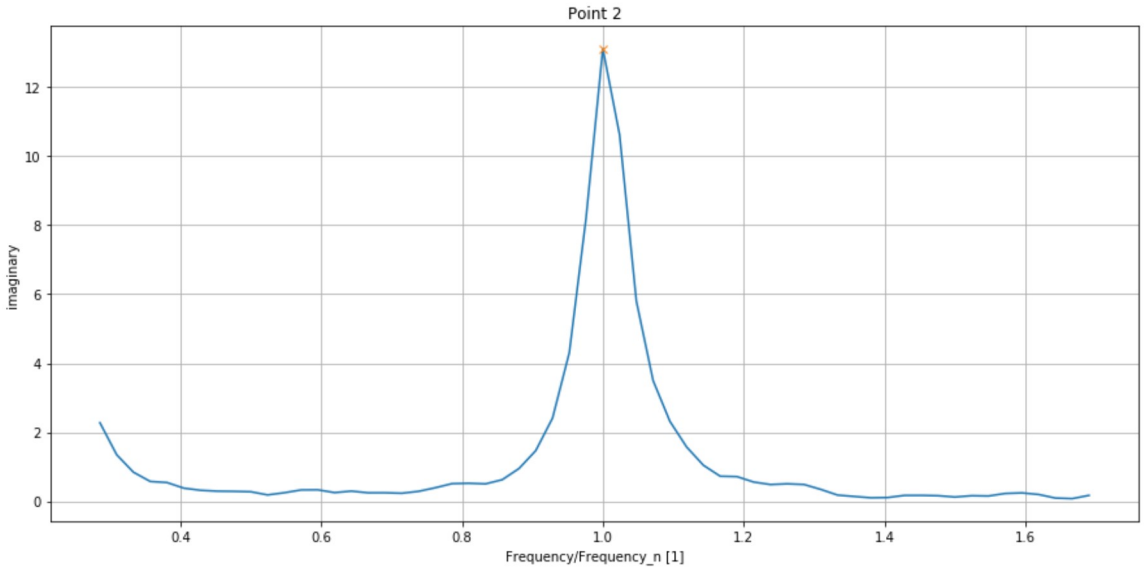

Point 3
real
Frequency/Frequency_n [1]

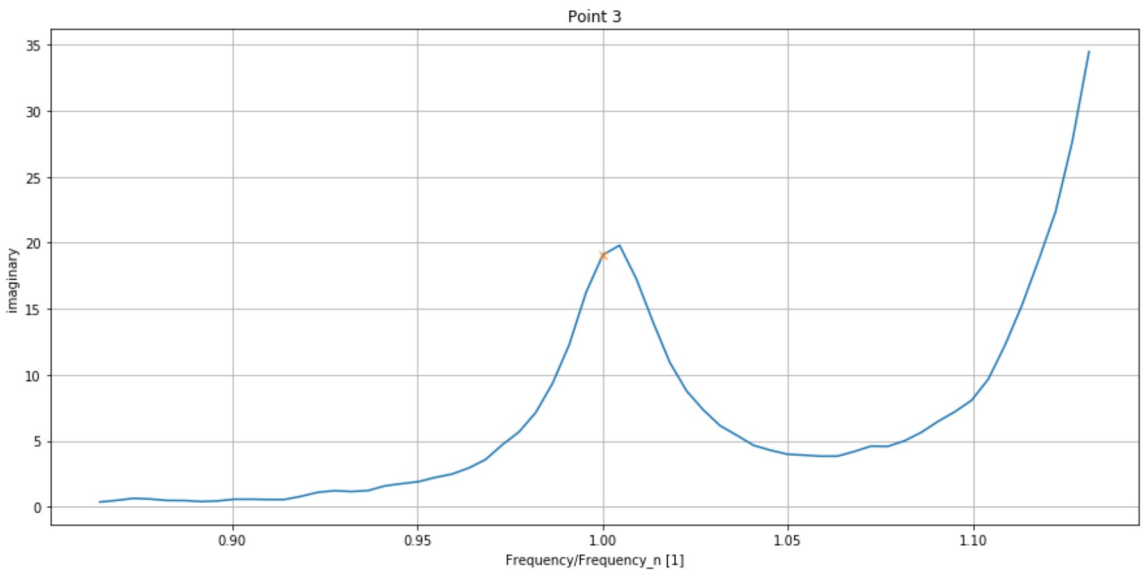

Point 4
real
Frequency/Frequency_n [1]

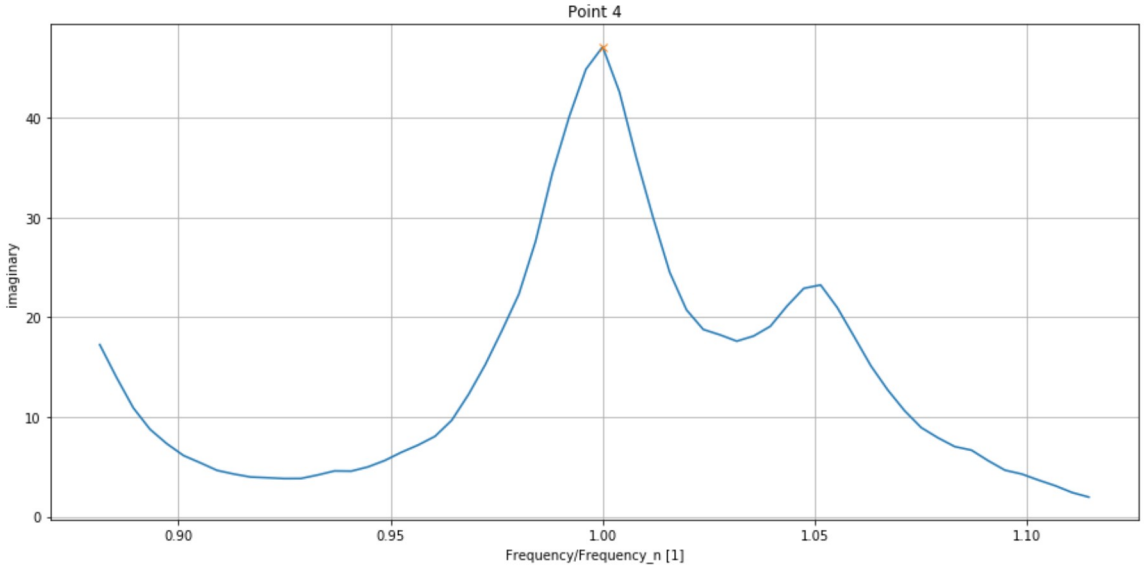

Point 5
real
Frequency/Frequency_n [1]

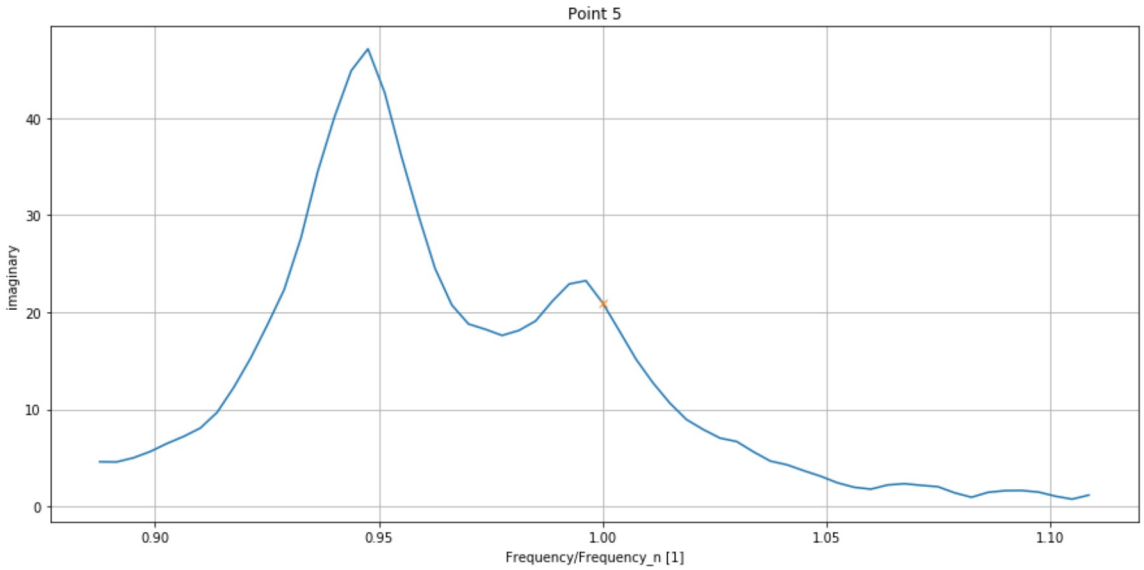
In [22]:
```
# Plot the imaginary part
for n,i in enumerate(peaks_idx_P05_rfft_window):
    plt.figure(figsize = [15,7])

    lower_freq = frequencies[i]-bound
    upper_freq = frequencies[i]+bound

    lower_idx = (np.abs(frequencies - lower_freq)).argmin()
    upper_idx = (np.abs(frequencies - upper_freq)).argmin()

    plt.plot(frequencies[lower_idx:upper_idx]/frequencies[i], np.imag(P05_recept
ance[lower_idx:upper_idx]))
    plt.plot(frequencies[i]/frequencies[i], np.imag(P05_receptance[i]), 'x')

    plt.ylabel('imaginary')
    plt.xlabel('Frequency/Frequency_n [1]')
    plt.title('Point '+ str(n+1))
    plt.grid()
```


Point 1
imaginary
Frequency/Frequency_n [1]


Point 2
imaginary
Frequency/Frequency_n [1]


Point 3
imaginary
Frequency/Frequency_n [1]

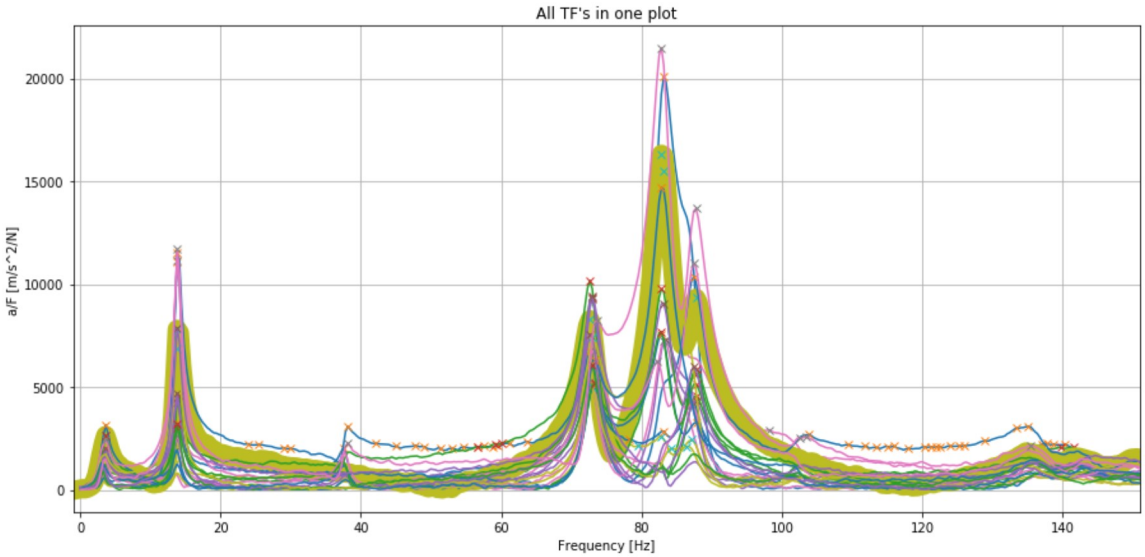




In [23]:
```
print('Peaks:', frequencies[peaks_idx_P05_rfft_window], 'Hz')
```

```
Peaks: [ 3.61604208 13.80670611 72.64957265 83.16896778 87.77120316] Hz
```

In [24]:
```
# All TF in one plot
plt.figure(figsize = [15,7])
plt.ylabel('a/F [m/s^2/N]')
plt.xlabel('Frequency [Hz]')
plt.xlim(-1,151)
plt.title('All TF\'s in one plot')
plt.grid()

for ii in range(2, 26):

    Pi_rfft_window = rfft(data[:,ii]*W)

    peaks_idx_Pi_receptance, _ = find_peaks(np.abs(Pi_rfft_window), height=2000)

    if ii == 6:
        plt.plot(frequencies, np.abs(Pi_rfft_window), linewidth=15)
        plt.plot(frequencies[peaks_idx_Pi_receptance], np.abs(Pi_rfft_window)[pe
aks_idx_Pi_receptance], 'x')
    else:
        plt.plot(frequencies, np.abs(Pi_rfft_window))
        plt.plot(frequencies[peaks_idx_Pi_receptance], np.abs(Pi_rfft_window)[pe
aks_idx_Pi_receptance], 'x')

plt.show()
```



Determine Damping Ratio

Determine the damping ratio of each natural frequency from the half power width. Use the drive point receptance.

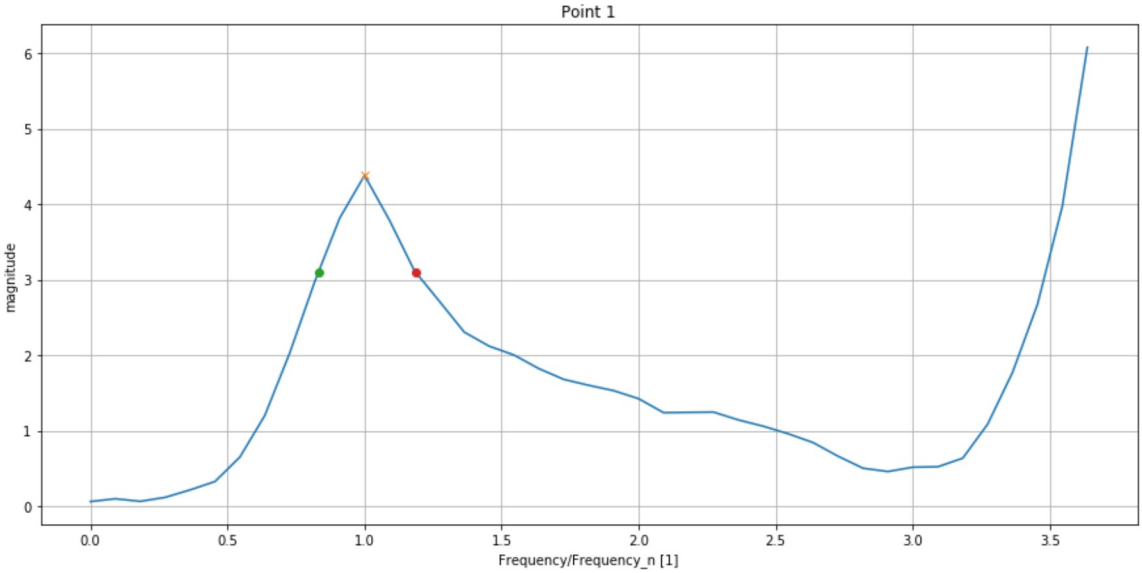
- plot the determined half power points on the transfer function for each peak

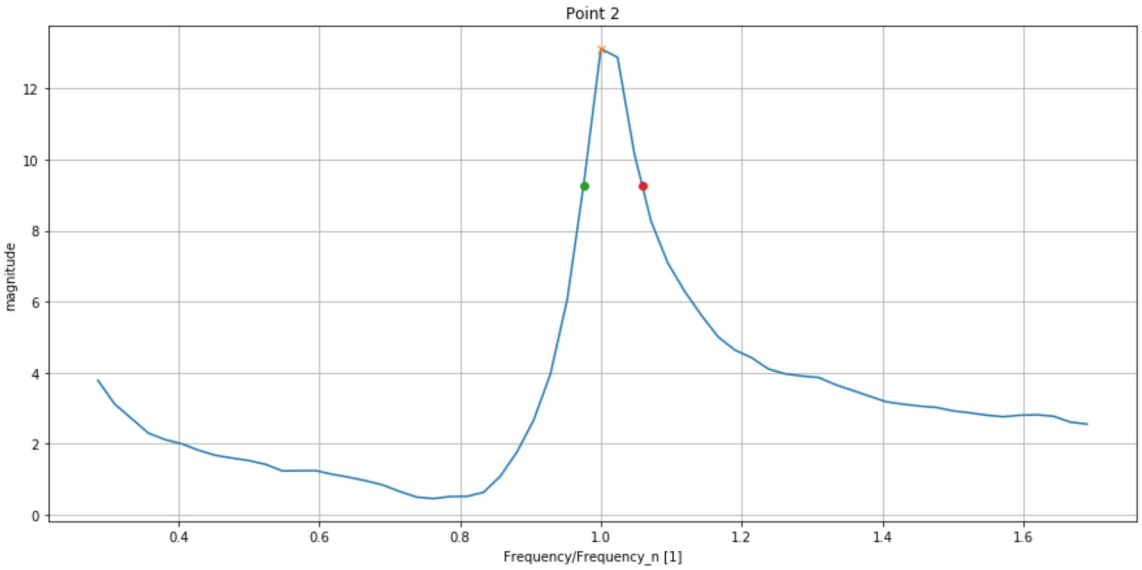
In [25]:
```
half_power = np.abs(P05_receptance)/sqrt(2) #
print('Half power magnitude:', half_power[peaks_idx_P05_rfft_window])
```

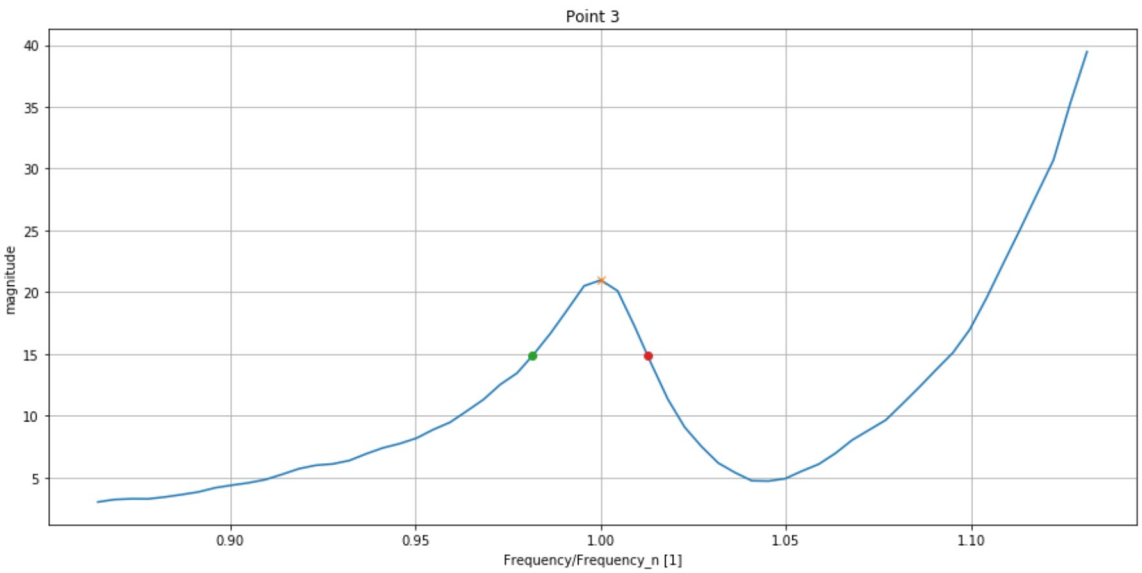
```
Half power magnitude: [ 3.09696453  9.27648256 14.83334402 33.35907903 20.3679
2258]
```

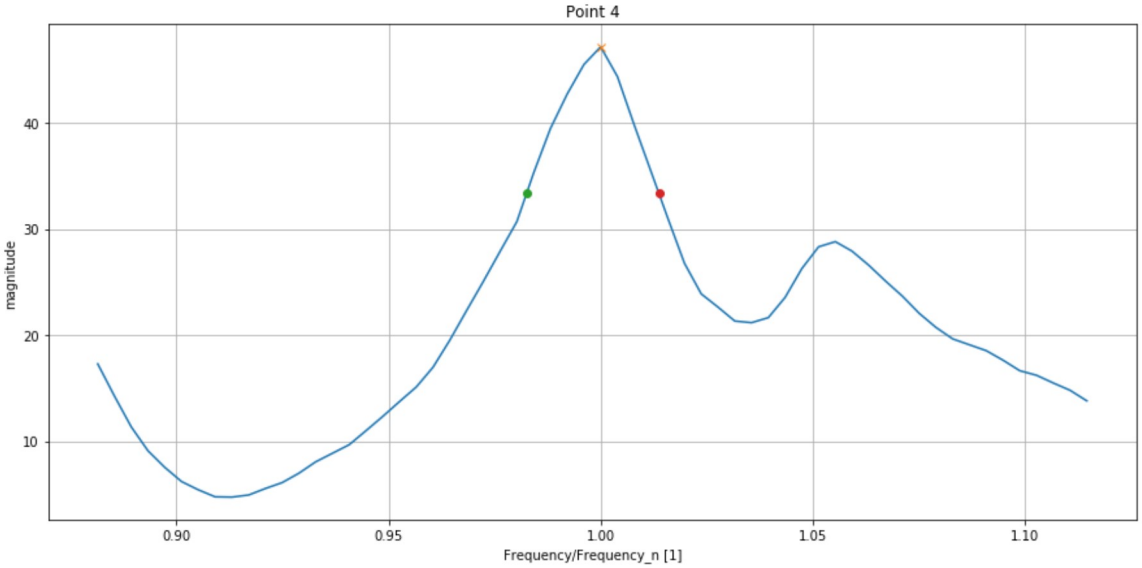
In [26]:
```
# find out to which index the half power points belong
index_half_power = np.zeros([len(peaks_idx_P05_rfft_window), 4],dtype = int)

for n,idx in enumerate(peaks_idx_P05_rfft_window):
    for i in range(idx, 0, -1):
        if half_power[idx] >= np.abs(P05_receptance)[i]:
            index_half_power[n,0] = i
            index_half_power[n,1] = i+1
            break

    for i in range(idx, len(frequencies)+1, 1):
        if half_power[idx] >= np.abs(P05_receptance)[i]:
            index_half_power[n,3] = i
            index_half_power[n,2] = i-1
            break

# linear interpolation between between the values found
freq_half_power = np.zeros([len(peaks_idx_P05_rfft_window),2])

for n,a in enumerate(index_half_power):
        freq_half_power[n,0] = ((frequencies[a[1]]-frequencies[a[0]])/
                                (np.abs(P05_receptance)[a[1]]-
                                 np.abs(P05_receptance)[a[0]]
                                ))*(half_power[peaks_idx_P05_rfft_window[n]]-np.
abs(P05_receptance)[a[0]])+frequencies[a[0]]
        freq_half_power[n,1] = ((frequencies[a[3]]-frequencies[a[2]])/
                                (np.abs(P05_receptance)[a[3]]-
                                 np.abs(P05_receptance)[a[2]]
                                ))*(half_power[peaks_idx_P05_rfft_window[n]]-np.
abs(P05_receptance)[a[2]])+frequencies[a[2]]
```

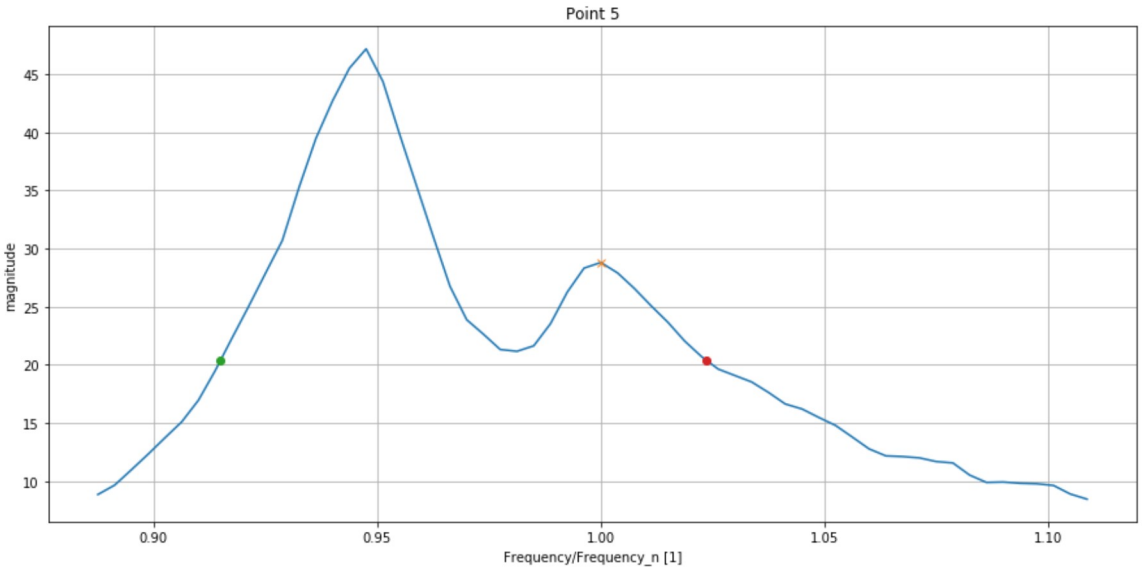
In [27]:
```
# Plot the magnitude
for n,i in enumerate(peaks_idx_P05_rfft_window):

    plt.figure(figsize = [15,7])

    lower_freq = frequencies[i]-bound
    upper_freq = frequencies[i]+bound

    lower_idx = (np.abs(frequencies - lower_freq)).argmin()
    upper_idx = (np.abs(frequencies - upper_freq)).argmin()

    if n == 4:
        P05_lower_idx = lower_idx
        P05_upper_idx = upper_idx
    else:
        pass

    plt.plot(frequencies[lower_idx:upper_idx]/frequencies[i], np.abs(P05_recepta
nce[lower_idx:upper_idx]))
    plt.plot(frequencies[i]/frequencies[i], np.abs(P05_receptance[i]), 'x')
    plt.plot(freq_half_power[n,0]/frequencies[i], half_power[i], 'o')
    plt.plot(freq_half_power[n,1]/frequencies[i], half_power[i], 'o')

    plt.ylabel('magnitude')
    plt.xlabel('Frequency/Frequency_n [1]')
    plt.title('Point '+ str(n+1))
    plt.grid()
```


Point 1
magnitude
Frequency/Frequency_n [1]


Point 2
magnitude
Frequency/Frequency_n [1]


Point 3
magnitude
Frequency/Frequency_n [1]



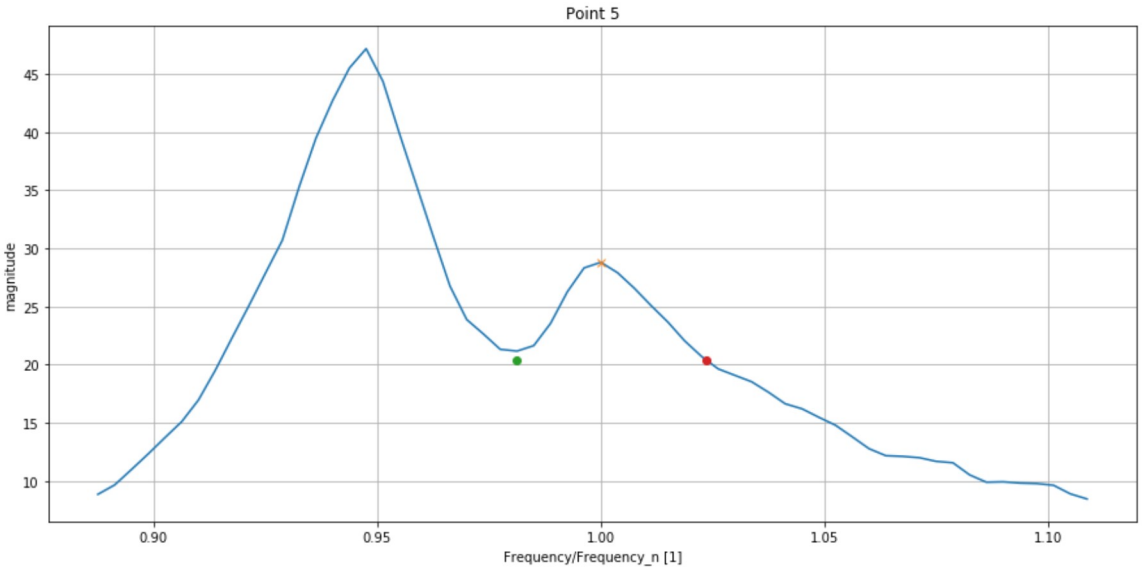


- what damping factors do you obtain?

In [28]:
```
# Damping factor
Damping_factor = (freq_half_power[:,1]-freq_half_power[:,0])/(2*frequencies[peak
s_idx_P05_rfft_window])

print('Damping Factors: ', Damping_factor)
```

```
Damping Factors:  [0.17744339 0.04160639 0.01560504 0.01559373 0.05433805]
```

In [29]:
```
# shift first half power of point 5 into valley
valley_idx = np.argmin(np.abs(P05_receptance[peaks_idx_P05_rfft_window[-2]:peaks
_idx_P05_rfft_window[-1]]))
freq_half_power[4,0] = frequencies[peaks_idx_P05_rfft_window[-2]:peaks_idx_P05_r
fft_window[-1]][valley_idx]

lower_freq = frequencies[peaks_idx_P05_rfft_window[-1]]-bound
upper_freq = frequencies[peaks_idx_P05_rfft_window[-1]]+bound

lower_idx = (np.abs(frequencies - lower_freq)).argmin()
upper_idx = (np.abs(frequencies - upper_freq)).argmin()

plt.figure(figsize = [15,7])
plt.plot(frequencies[lower_idx:upper_idx]/frequencies[peaks_idx_P05_rfft_window
[-1]],
         np.abs(P05_receptance[lower_idx:upper_idx]))
plt.plot(frequencies[peaks_idx_P05_rfft_window[-1]]/frequencies[peaks_idx_P05_rf
ft_window[-1]],
         np.abs(P05_receptance[peaks_idx_P05_rfft_window[-1]]), 'x')
plt.plot(freq_half_power[4,0]/frequencies[peaks_idx_P05_rfft_window[-1]], half_p
ower[peaks_idx_P05_rfft_window[-1]], 'o')
plt.plot(freq_half_power[4,1]/frequencies[peaks_idx_P05_rfft_window[-1]], half_p
ower[peaks_idx_P05_rfft_window[-1]], 'o')

plt.ylabel('magnitude')
plt.xlabel('Frequency/Frequency_n [1]')
plt.title('Point '+ str(5))
plt.grid()
```



In [30]:
```
# Damping factors after shifting point 5
Damping_factor = (freq_half_power[:,1]-freq_half_power[:,0])/(2*frequencies[peak
s_idx_P05_rfft_window])

print('Damping Factors: ', Damping_factor)
```

```
Damping Factors:  [0.17744339 0.04160639 0.01560504 0.01559373 0.02120498]
```

Circle Fitting

Employ the circle fitting algorithm to determine natural frequencies and damping factors.

- Develop a function that fits a circle in the nyquist plane to given data points (receptance values and corresponding frequencies). It should return the natural frequency, damping factor and complex valued modal constant.

In [31]:
```
def CircleFitting(receptance,f,printFlag = 0) :
    x = np.real(receptance)
    y = np.imag(receptance)
    w = 2*np.pi*f
    dw = w[1] - w[0]

    ## DO CIRCLE FITTING STUFF TO THE DATA
    L = len(x)
    A = np.zeros((3,3))

    A[0,0] = np.sum(x*x)
    A[1,1] = np.sum(y*y)
    A[2,2] = L

    A[0,1] = A[1,0] = np.sum(x*y)
    A[0,2] = A[2,0] = -np.sum(x)
    A[1,2] = A[2,1] = -np.sum(y)

    B = np.zeros((3,1))
    B[0] = -np.sum(x*x*x)-np.sum(x*y*y)
    B[1] = -np.sum(y*y*y)-np.sum(x*x*y)
    B[2] = np.sum(x*x)+np.sum(y*y)

    sol = np.linalg.solve(A,B)

    ## CIRCLE DATA
    xk = -1/2*sol[0]
    yk = -1/2*sol[1]
    Rk = np.sqrt(sol[2] + xk**2 + yk**2)

    # Generate spokes for plotting
    pointers_x = x - xk
    pointers_y = y - yk

    frequency_spacing = np.zeros((len(pointers_x)-1))

    ## CALCULATE ANGLES BETWEEN SPOKES
    for i in range(0,len(pointers_x)-1):
        pointer_1 = np.array((pointers_x[i],pointers_y[i]))
        pointer_2 = np.array((pointers_x[i+1],pointers_y[i+1]))
        frequency_spacing[i] = np.arccos(np.dot(pointer_1,pointer_2)/
                                         np.linalg.norm(pointer_1)/
                                         np.linalg.norm(pointer_2))

    ## CALCULATE GRADIENT
    frequency_spacing = np.cumsum(frequency_spacing)
    dg_dw = np.gradient(frequency_spacing,dw)
    d2g_dw2 = np.gradient(dg_dw,dw)
    idx_max = np.argmax(dg_dw)

    # Find Maximum and Minium Values of the function --> somewhere between them
we have the 0 crossing
    max_idx_d2g_dw2 = (np.argmax(d2g_dw2))
    min_idx_d2g_dw2 = (np.argmin(d2g_dw2))

    # Flip values of d2g_dw2, need the function to be increasing ---> form - val
ues to + values
    x_values = np.flip(d2g_dw2[max_idx_d2g_dw2:min_idx_d2g_dw2+1])
    y_values = np.flip(f[max_idx_d2g_dw2:min_idx_d2g_dw2+1])

    # Find zero values
    freq_zero_crossing = np.interp(0.0,x_values,y_values)

    # Find x,y values of wk on given data
    x_wk = np.interp(freq_zero_crossing,f,x)
    y_wk = np.interp(freq_zero_crossing,f,y)

    ## FIND wk
```

- First test your function on a nice, prominant peak of the drive point receptance (e.g. around 73Hz): Select suitable data points (frequency range) for the fit. Plot your fit in the Nyquist plane. Also plot the transfer function estimation by the cicle fit, i.e.

$$H(\omega) = \frac{C_k}{\omega_k^2 - \omega^2 + 2j\omega_k\zeta_k\omega} + D_k$$

  in a wider frequency range, and compare it to the data to see how the fit performs.
- Use your function and determine suitable frequency ranges to fit all natural frequencies. For some it might be useful to switch to a different transfer function, where the peak under consideration is more prominant.

The plotting of HW on a wider range is done in the last point below.

Range for Circle Fitting:

- Point 1
  - [0,15] -> 1,2,7,8,23,24 measurement point
  - [0,31] -> rest
- Point 2
  - [32,70] -> 3,9,10,15,21,22
  - [32,150] -> rest
- Point 3
  - [151,230] -> rest !Caution! Point 24
- Point 4
  - [240,262] -> 2,3,6,7,10,11,14,15,18,19,22
  - [231,262] -> rest
  - [240,255] -> 23
- Point 5
  - [262,280] -> 3,22
  - [262,350] -> rest

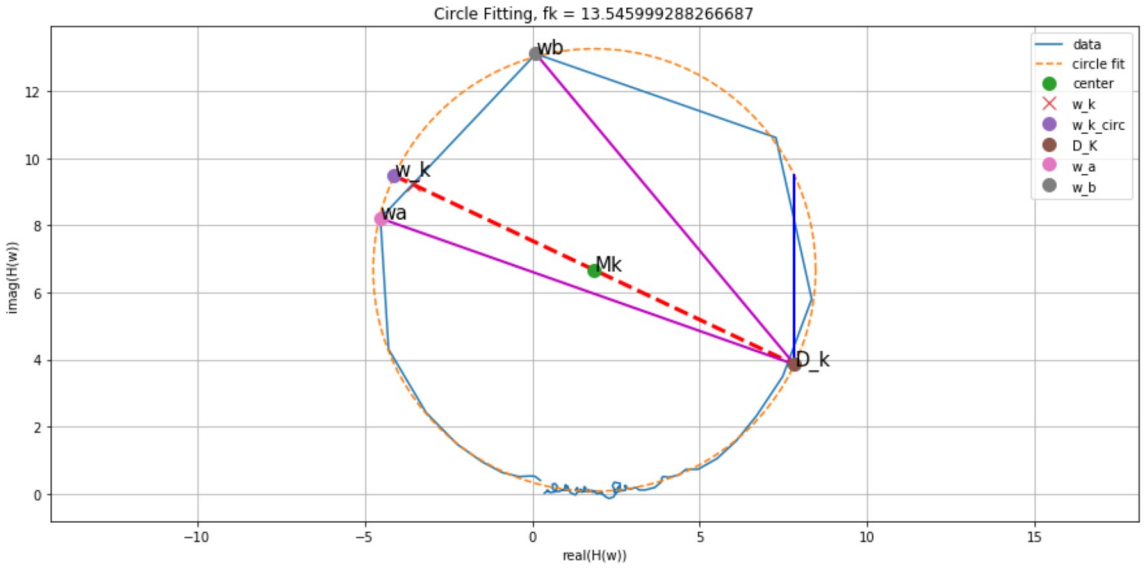
The one generated for only P05 made problems with with some points, just for reference

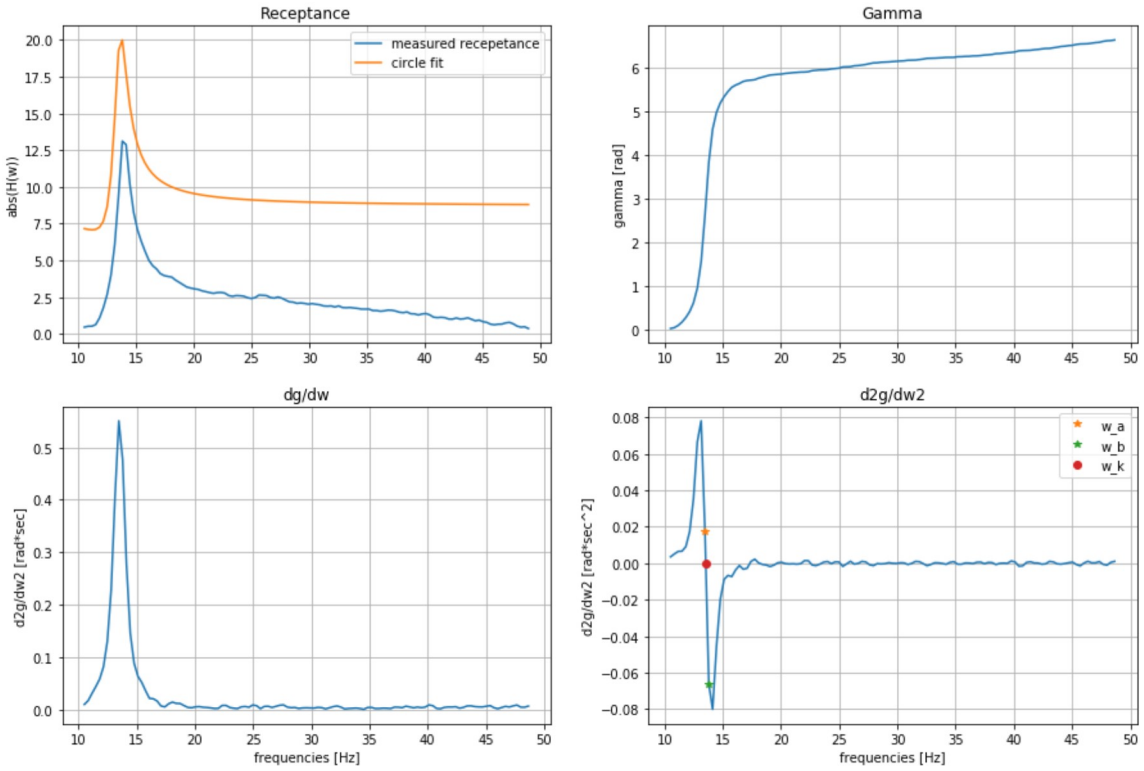
```
frequency_indexes = np.array([[0,31],
                              [32,150],
                              [151,230],
                              [231,262],
                              [262,350]
                              ])
```

In [32]:
```
# # ------------------------------ USED TO DEBUG AND FIND RANGE FOR CIRCLE FITTI
NG ------------------------------------
# window = range(240,255)
# plt.figure(figsize = [15,50])

# for i in range(2,26):
#     Pi_receptance = rfft(data[:,i]*W)/rfft(data[:,1]*W)

#     plt.subplot(13,2,i-1)
#     plt.title('Plot for measurementpoint: ' + str(i-1))
#     plt.grid()
#     plt.plot(np.abs(Pi_receptance[window]))
# plt.show()
```

In [33]:
```
# ------------------------------ USED TO DEBUG AND FIND RANGE FOR CIRCLE FITTING
-------------------------------------
# window = range(262,280)
# plt.figure(figsize = [15,7])

# i = 3
# Pi_receptance = rfft(data[:,i + 1]*W)/rfft(data[:,1]*W)

# plt.title('Plot for measurementpoint: ' + str(i))
# plt.grid()
# plt.plot(np.abs(Pi_receptance[window]))
# plt.show()
```

In [34]:
```
## EXAMPLE OF THE CIRCLE FITTING TOOL
lower_idx = 32
upper_idx = 150

f = frequencies[lower_idx:upper_idx]
receptance = P05_receptance[lower_idx:upper_idx]

# Toggle plots ON/Off with 0 or 1 as flag in Circle Fitting
[wk, zeta_k, C_k, D_k, H_w] = CircleFitting(receptance,f,1)

print("------------------------- Peak -------------------------")
print("Natural Frequency: ", wk/2/np.pi)
print("Damping Factor: ", zeta_k)
print("Modal Constant: ", C_k)
print()
```






```
------------------------- Peak -------------------------
Natural Frequency:  13.545999288266687
Damping Factor:  -0.04308529449229031
Modal Constant:  (-3516.5746148786898-7457.778440193097j)
```

- Use the circle fit algorithm to determine the modal constants for all measured locations (=recpetance functions). Use the frequency ranges (around the peaks) determined above.

In [35]:
```
# Build some containers to save all the fittings
size_container = (len(peaks_idx_P05_rfft_window),24)

wk_all = np.zeros(size_container, dtype = float)
zeta_k_all = np.zeros(size_container, dtype = float)
C_k_all = np.zeros(size_container, dtype = complex)
D_k_all = np.zeros(size_container, dtype = complex)
```

In [36]:
```
# Lets do it for the first point
# ------------------------------
#     - [0,15] -> 1,2,7,8,23,24
#     - [0,31] -> rest
indexi = np.array([1,2,7,8,23,24])
k = 0

for i in range(1,25):
    if i in indexi:
        fRange = range(0,15)
    else:
        fRange = range(0,31)

    Pi_receptance = rfft(data[:,i+1]*W)/rfft(data[:,1]*W)
    f = frequencies[fRange]
    receptance = Pi_receptance[fRange]

    # Toggle plots ON/Off with 0 or 1 as flag in Circle Fitting
    [wk, zeta_k, C_k, D_k, H_w] = CircleFitting(receptance,f,0)

    wk_all[k,i-1] = wk
    zeta_k_all[k,i-1] = zeta_k
    C_k_all[k,i-1] = C_k
    D_k_all[k,i-1] = D_k

#     print("------------------------ Point " + str(i) +
"-------------------------")
#     print("Natural Frequency: ", wk/2/np.pi)
#     print("Damping Factor: ", zeta_k)
#     print("Modal Constant: ", C_k)
#     print()
```

In [37]:
```
# Lets do it for the Second point
# ------------------------------
#     - [31,70] -> 3,9,10,15,21,22
#     - [31,150] -> rest
indexi = np.array([3,9,10,15,21,22])
k = 1

for i in range(1,25):
    if i in indexi:
        fRange = range(31,70)
    else:
        fRange = range(31,150)

    Pi_receptance = rfft(data[:,i+1]*W)/rfft(data[:,1]*W)
    f = frequencies[fRange]
    receptance = Pi_receptance[fRange]

    # Toggle plots ON/Off with 0 or 1 as flag in Circle Fitting
    [wk, zeta_k, C_k, D_k, H_w] = CircleFitting(receptance,f,0)

    wk_all[k,i-1] = wk
    zeta_k_all[k,i-1] = zeta_k
    C_k_all[k,i-1] = C_k
    D_k_all[k,i-1] = D_k

#     print("------------------------- Point " + str(i) +
"-------------------------")
#     print("Natural Frequency: ", wk/2/np.pi)
#     print("Damping Factor: ", zeta_k)
#     print("Modal Constant: ", C_k)
#     print()
```

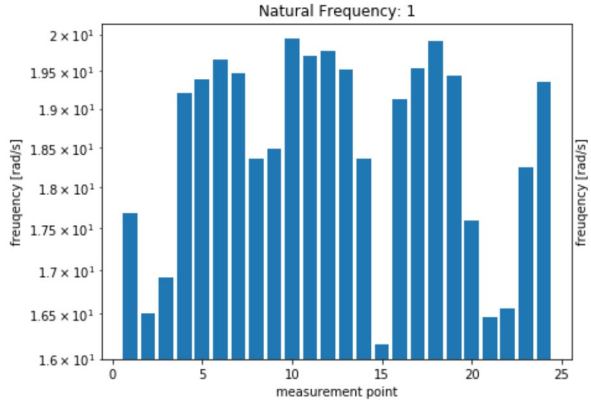
In [38]:
```
# Lets do it for the Third point
# ------------------------------
# [151,230] -> all
fRange = range(151,230)
k = 2

for i in range(1,25):
    Pi_receptance = rfft(data[:,i+1]*W)/rfft(data[:,1]*W)
    f = frequencies[fRange]
    receptance = Pi_receptance[fRange]

    # Toggle plots ON/Off with 0 or 1 as flag in Circle Fitting
    [wk, zeta_k, C_k, D_k, H_w] = CircleFitting(receptance,f,0)

    wk_all[k,i-1] = wk
    zeta_k_all[k,i-1] = zeta_k
    C_k_all[k,i-1] = C_k
    D_k_all[k,i-1] = D_k

#     print("------------------------- Point " + str(i) +
"-------------------------")
#     print("Natural Frequency: ", wk/2/np.pi)
#     print("Damping Factor: ", zeta_k)
#     print("Modal Constant: ", C_k)
#     print()
```

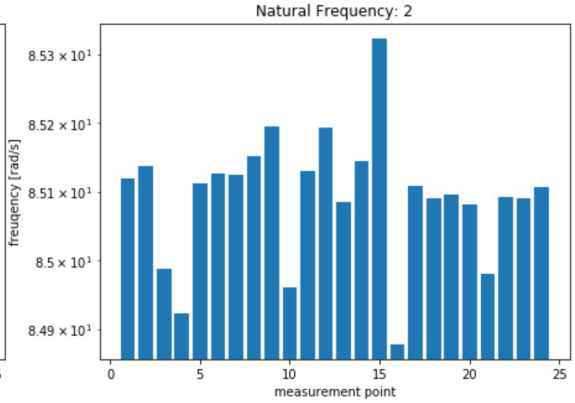
In [39]:
```
# Lets do it for the Fourth point
# ------------------------------
# [240,262] -> 2,3,6,7,10,11,14,15,18,19,22
# [231,262] -> rest
# [240,255] -> 23
indexi = np.array([2,3,6,7,10,11,14,15,18,19,22])
indexii = np.array([23])
k = 3

for i in range(1,25):
    if i in indexi:
        fRange = range(240,262)
    elif i in indexii:
        fRange = range(240,255)
    else:
        fRange = range(231,262)

    Pi_receptance = rfft(data[:,i+1]*W)/rfft(data[:,1]*W)
    f = frequencies[fRange]
    receptance = Pi_receptance[fRange]

    # Toggle plots ON/Off with 0 or 1 as flag in Circle Fitting
    [wk, zeta_k, C_k, D_k, H_w] = CircleFitting(receptance,f,0)

    wk_all[k,i-1] = wk
    zeta_k_all[k,i-1] = zeta_k
    C_k_all[k,i-1] = C_k
    D_k_all[k,i-1] = D_k

#     print("------------------------- Point " + str(i) +
"-------------------------")
#     print("Natural Frequency: ", wk/2/np.pi)
#     print("Damping Factor: ", zeta_k)
#     print("Modal Constant: ", C_k)
#     print()
```

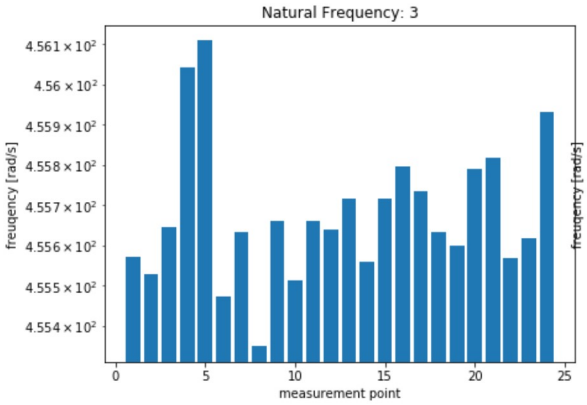
In [40]:
```
# Lets do it for the fifth point
# ------------------------------
#     - [262,280] -> 3,22
#     - [262,350] -> rest
indexi = np.array([3,22])
k = 4

for i in range(1,25):
    if i in indexi:
        fRange = range(262,280)
    else:
        fRange = range(262,350)

    Pi_receptance = rfft(data[:,i+1]*W)/rfft(data[:,1]*W)
    f = frequencies[fRange]
    receptance = Pi_receptance[fRange]

    # Toggle plots ON/Off with 0 or 1 as flag in Circle Fitting
    [wk, zeta_k, C_k, D_k, H_w] = CircleFitting(receptance,f,0)

    wk_all[k,i-1] = wk
    zeta_k_all[k,i-1] = zeta_k
    C_k_all[k,i-1] = C_k
    D_k_all[k,i-1] = D_k

#     print("------------------------- Point " + str(i) +
"-------------------------")
#     print("Natural Frequency: ", wk/2/np.pi)
#     print("Damping Factor: ", zeta_k)
#     print("Modal Constant: ", C_k)
#     print()
```

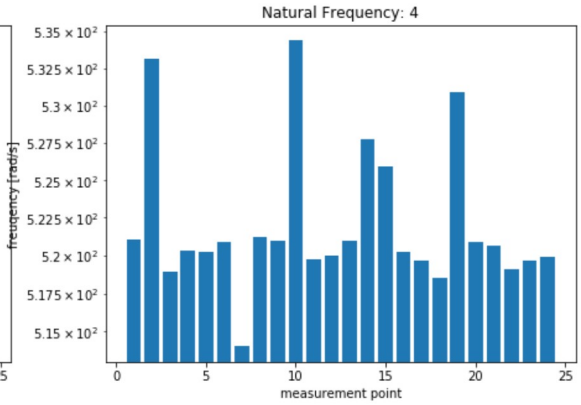
- Do the natural frequecies and damping factors obtained from different transfer functions differ?

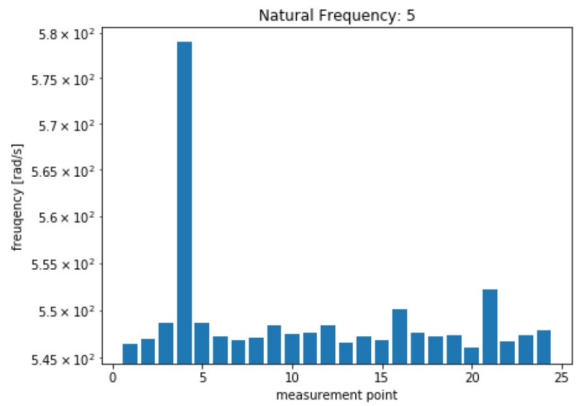
In [41]:
```
# Plot wks
index = np.arange(1,25)

plt.figure(figsize = [15,17])

plt.subplot(3,2,1)
plt.bar(index, wk_all[0,:])
plt.xlabel('measurement point')
plt.ylabel('freuqency [rad/s]')
plt.title('Natural Frequency: 1')
plt.yscale('log')

plt.subplot(3,2,2)
plt.bar(index, wk_all[1,:])
plt.xlabel('measurement point')
plt.ylabel('freuqency [rad/s]')
plt.title('Natural Frequency: 2')
plt.yscale('log')

plt.subplot(3,2,3)
plt.bar(index, wk_all[2,:])
plt.xlabel('measurement point')
plt.ylabel('freuqency [rad/s]')
plt.title('Natural Frequency: 3')
plt.yscale('log')

plt.subplot(3,2,4)
plt.bar(index, wk_all[3,:])
plt.xlabel('measurement point')
plt.ylabel('freuqency [rad/s]')
plt.title('Natural Frequency: 4')
plt.yscale('log')

plt.subplot(3,2,5)
plt.bar(index, wk_all[4,:])
plt.xlabel('measurement point')
plt.ylabel('freuqency [rad/s]')
plt.title('Natural Frequency: 5')
plt.yscale('log')

plt.show()
```


Natural Frequency: 1
freuqency [rad/s]
measurement point

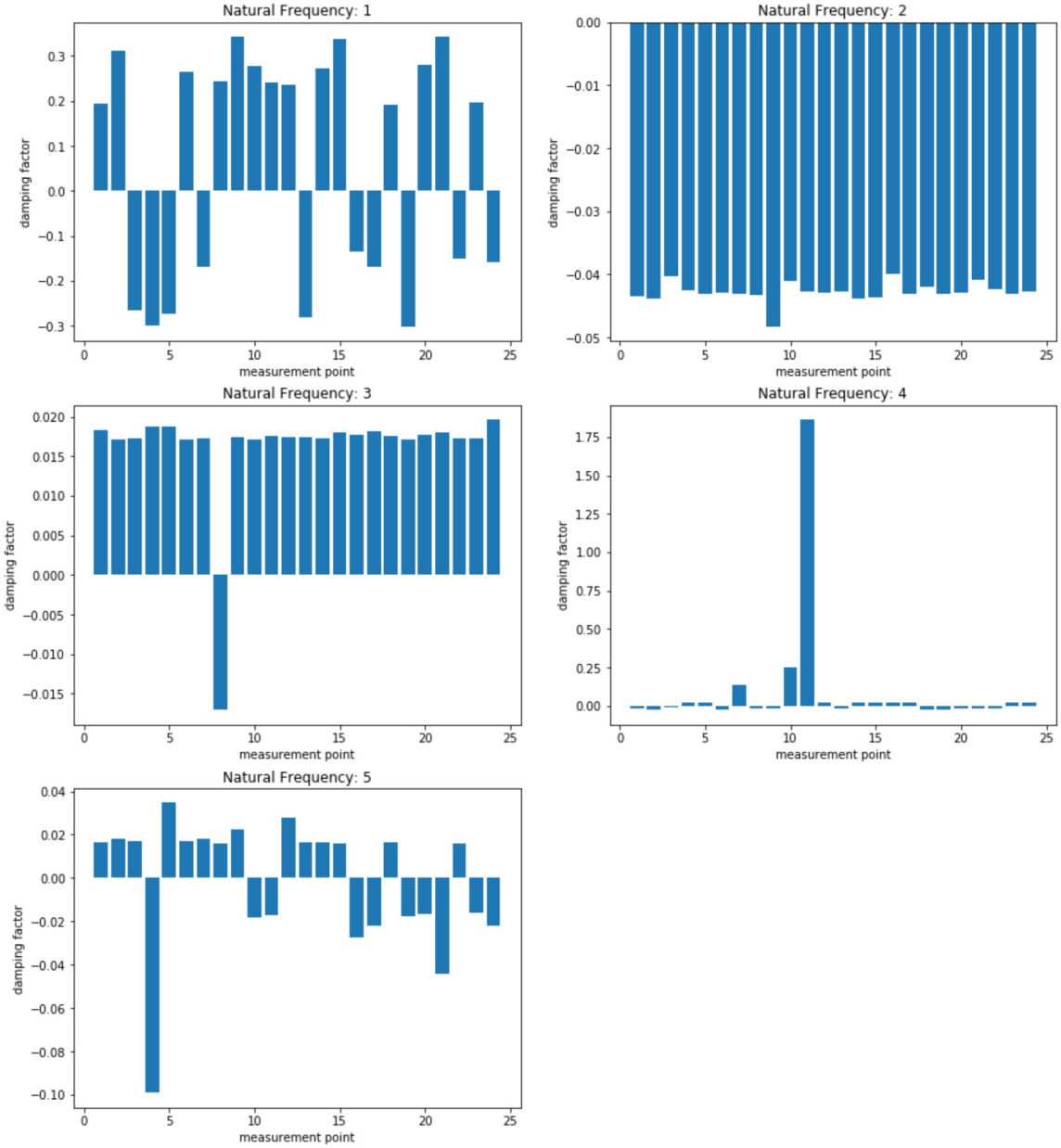

Natural Frequency: 2
freuqency [rad/s]
measurement point


Natural Frequency: 3
freuqency [rad/s]
measurement point


Natural Frequency: 4
freuqency [rad/s]
measurement point

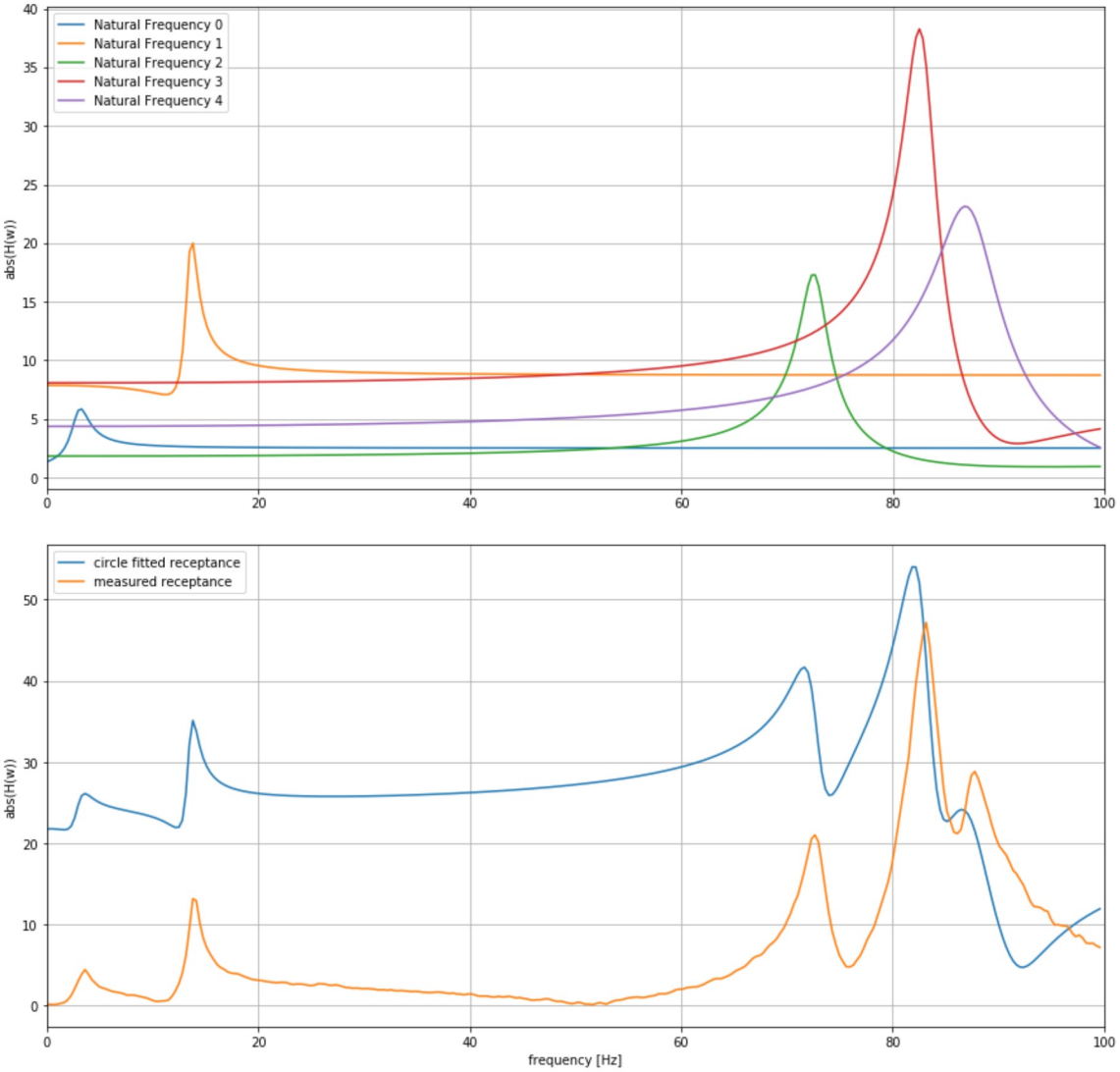

Natural Frequency: 5
freuqency [rad/s]
measurement point

In [42]:
```
# Plot damping factors
index = np.arange(1,25)

plt.figure(figsize = [15,17])

plt.subplot(3,2,1)
plt.bar(index, zeta_k_all[0,:])
plt.xlabel('measurement point')
plt.ylabel('damping factor')
plt.title('Natural Frequency: 1')
# plt.yscale('log')

plt.subplot(3,2,2)
plt.bar(index, zeta_k_all[1,:])
plt.xlabel('measurement point')
plt.ylabel('damping factor')
plt.title('Natural Frequency: 2')
# plt.yscale('log')

plt.subplot(3,2,3)
plt.bar(index, zeta_k_all[2,:])
plt.xlabel('measurement point')
plt.ylabel('damping factor')
plt.title('Natural Frequency: 3')
# plt.yscale('log')

plt.subplot(3,2,4)
plt.bar(index, zeta_k_all[3,:])
plt.xlabel('measurement point')
plt.ylabel('damping factor')
plt.title('Natural Frequency: 4')
# plt.yscale('log')

plt.subplot(3,2,5)
plt.bar(index, zeta_k_all[4,:])
plt.xlabel('measurement point')
plt.ylabel('damping factor')
plt.title('Natural Frequency: 5')
# plt.yscale('log')

plt.show()
```



- Use the moda parameters obtained via the circle fit to estimate the drive point recepance. Plot the measured repeptance, the single dregree of freedom estimates from the circle fit, as well as their sum (all in one plot to compare).

In [43]:
```
# Drive-Point --> P05 --> index 4
max_idx = np.abs(frequencies-100).argmin();
w = frequencies[0:max_idx]*2*np.pi
H_w_sum = np.zeros_like(frequencies[0:max_idx], dtype = 'complex')

plt.figure(figsize = [15,7])
for i, wk in enumerate(wk_all[:,4]):
    H_w = C_k_all[i,4]/(wk**2 - w*w + 2*1j*wk*zeta_k_all[i,4]*w) +  D_k_all[i,4]
    H_w_sum = H_w_sum + H_w

    plt.plot(frequencies[0:max_idx],np.abs(H_w),label = 'Natural Frequency ' + s
tr(i))
plt.xlim([0,100])
plt.ylabel('abs(H(w))')
plt.grid()
plt.legend()
plt.show()

# Plot Measurement and sum of ours
# shift = 22
shift = 0
plt.figure(figsize = [15,7])
plt.plot(frequencies[0:max_idx],np.abs(H_w_sum)-shift, label = 'circle fitted re
ceptance')
plt.plot(frequencies[0:max_idx],np.abs(P05_receptance[0:max_idx]), label = 'meas
ured receptance')
plt.xlim([0,100])
plt.grid()
plt.ylabel('abs(H(w))')
plt.xlabel('frequency [Hz]')
plt.legend()
plt.show()
```



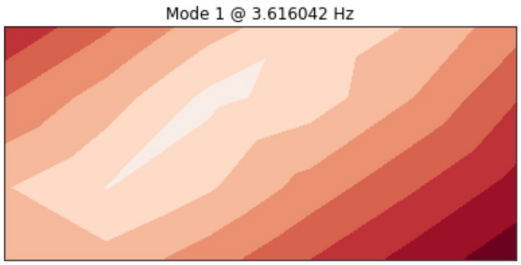
Task 2: Compute the Mode Shapes

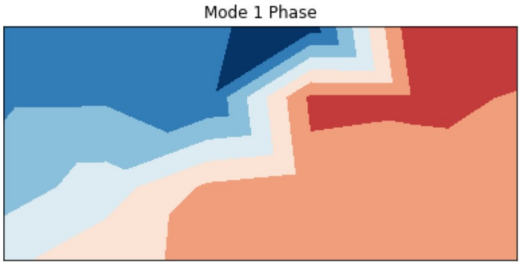
Having obtained natural frequencies and damping factors, from peak-picking (maximum maplitude, quadrature component, ...) use the recepance curves to determine the modal constants and mode shapes. The mode shapes will be complex valued in ganeral (due to the damping present in the system). As the are complex-valued you can display them as real/imaginary part, or absolute value and phase. Try both!

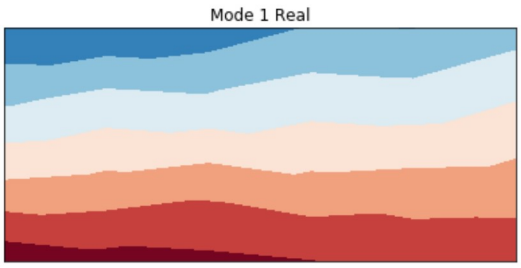
In [44]:
```
def plotmode2d(v,x,y,lim=None) :
    """plots a mode v defined at the points (x,y)"""
    if lim==None:
        lim = np.max(np.abs(v))
    plt.tricontourf(x,y,v,cmap=plt.get_cmap('RdBu_r'),vmin=-lim,vmax=lim)
#     plt.set_aspect('equal')
    plt.xticks([])
    plt.yticks([])
    #fig.tight_layout()
```

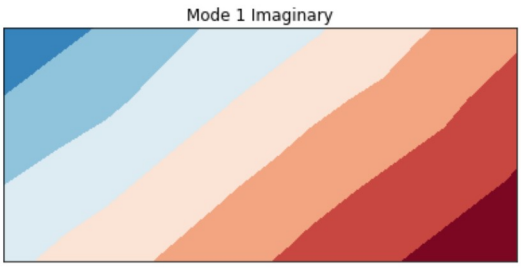
- Compute and plot the mode shapes from the peak-picking procedure

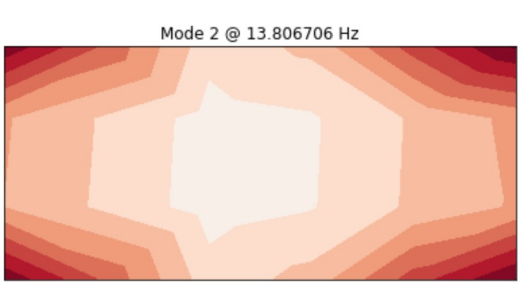
In [45]:
```
coordinates = loadtxt('measurement-coordinates.txt')

Mode_shapes_peakpicking = np.zeros((24,5),dtype= complex)

for k in range(0,5):
    H_ai = np.zeros(24, dtype = 'complex')
    zeta_k = Damping_factor[k]
    wk = 2*np.pi*frequencies[peaks_idx_P05_rfft_window[k]]

    for i in range(0,24):
        a = rfft(data[:,i+2]*W)/rfft(data[:,1]*W)
        H_ai[i] = a[peaks_idx_P05_rfft_window[k]]

    C = H_ai*2*1j*wk**2

    vak = np.sqrt(2*1j*zeta_k*wk**2*P05_receptance[peaks_idx_P05_rfft_window
[k]])

    vbk = C/vak
    Mode_shapes_peakpicking[:,k]= vbk

    plt.figure(figsize=[15,7])
    plt.subplot(2,2,1)
    plotmode2d(np.abs(vbk),coordinates[:,1],coordinates[:,2])
    plt.title('Mode %i @ %f Hz'%(k+1,frequencies[peaks_idx_P05_rfft_window[k]]))

    plt.subplot(2,2,2)
    plotmode2d(np.angle(vbk),coordinates[:,1],coordinates[:,2])
    plt.title('Mode %i Phase'%(k+1))

    plt.subplot(2,2,3)
    plotmode2d(np.real(vbk),coordinates[:,1],coordinates[:,2])
    plt.title('Mode %i Real'%(k+1))

    plt.subplot(2,2,4)
    plotmode2d(np.imag(vbk),coordinates[:,1],coordinates[:,2])
    plt.title('Mode %i Imaginary'%(k+1))
```

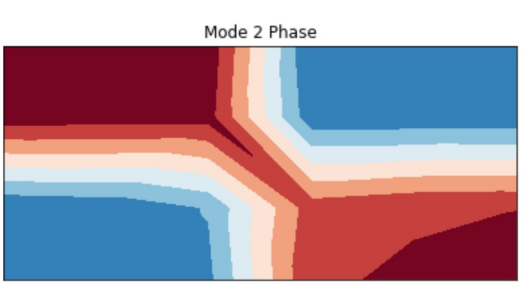

Mode 1 @ 3.616042 Hz

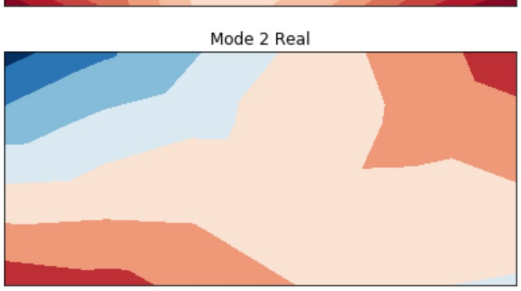

Mode 1 Phase

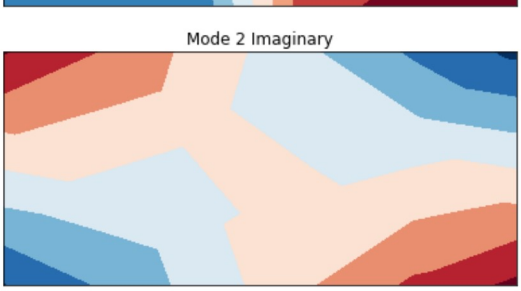

Mode 1 Real

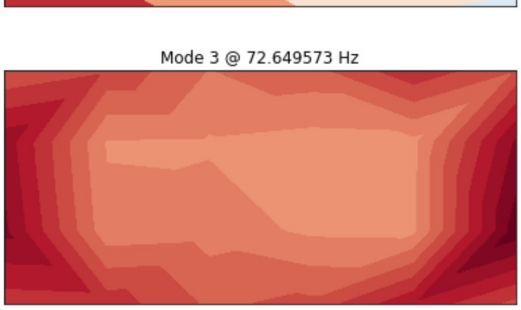

Mode 1 Imaginary

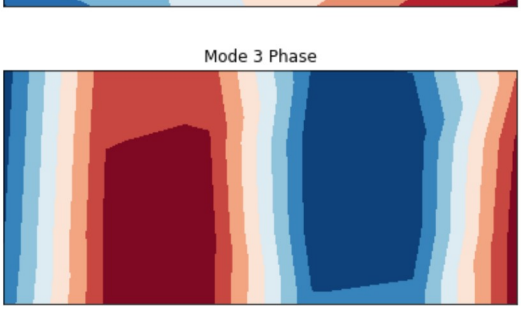

Mode 2 @ 13.806706 Hz

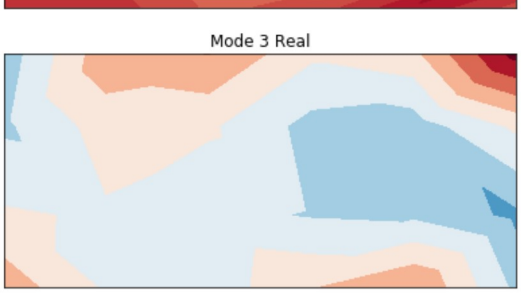

Mode 2 Phase

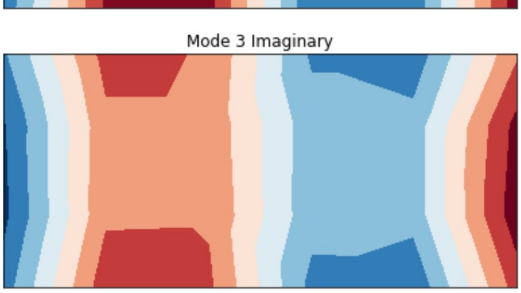

Mode 2 Real

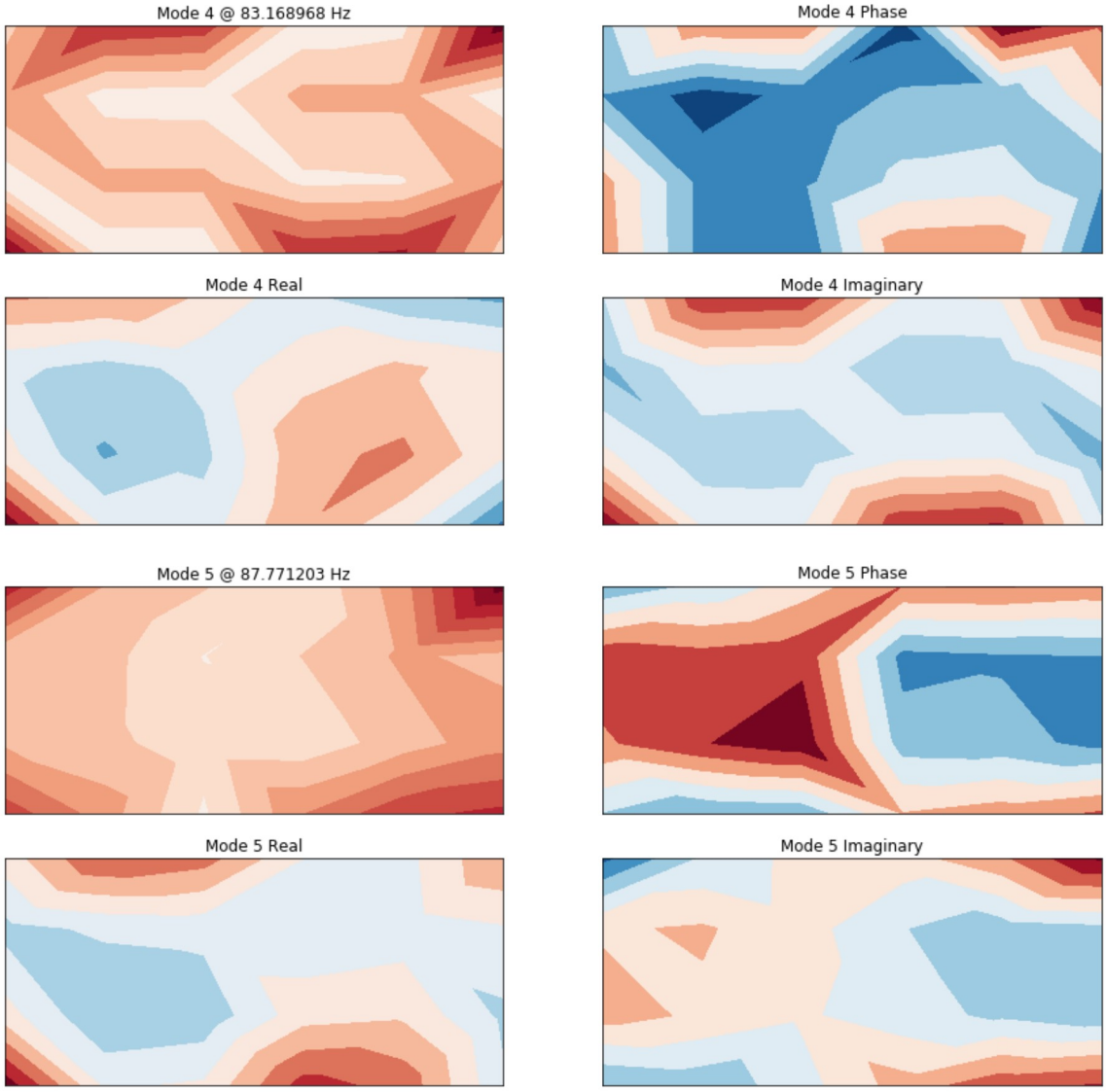

Mode 2 Imaginary


Mode 3 @ 72.649573 Hz


Mode 3 Phase


Mode 3 Real

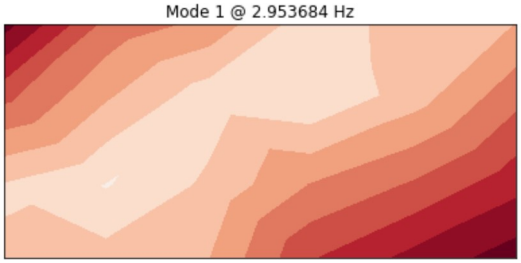

Mode 3 Imaginary



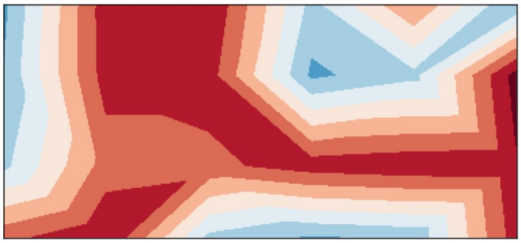
- Compute and plot the modes from circle fitting

In [46]:
```
Mode_shapes_circle = np.zeros((24,5),dtype = complex)

for i in range(0,5):
    vbk_circle = C_k_all[i,:]/np.sqrt(C_k_all[i,4])
    Mode_shapes_circle[:,i]= vbk_circle

    plt.figure(figsize=[15,7])
    plt.subplot(2,2,1)
    plotmode2d(np.abs(vbk_circle),coordinates[:,1],coordinates[:,2])
    plt.title('Mode %i @ %f Hz'%(i+1,np.mean(wk_all[i,:]/2/pi)))

    plt.subplot(2,2,2)
    plotmode2d(np.angle(vbk_circle),coordinates[:,1],coordinates[:,2])
    plt.title('Mode %i Phase'%(i+1))

    plt.subplot(2,2,3)
    plotmode2d(np.real(vbk_circle),coordinates[:,1],coordinates[:,2])
    plt.title('Mode %i Real'%(i+1))

    plt.subplot(2,2,4)
    plotmode2d(np.imag(vbk_circle),coordinates[:,1],coordinates[:,2])
    plt.title('Mode %i Imaginary'%(i+1))
```

Mode 1 @ 2.953684 Hz

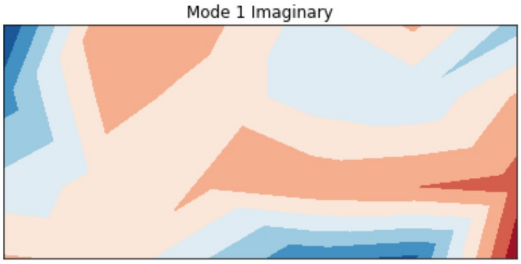


Mode 1 Phase

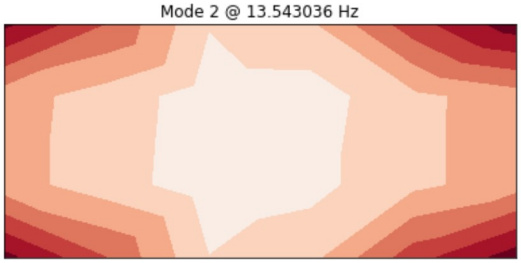


Mode 1 Real

Mode 1 Imaginary

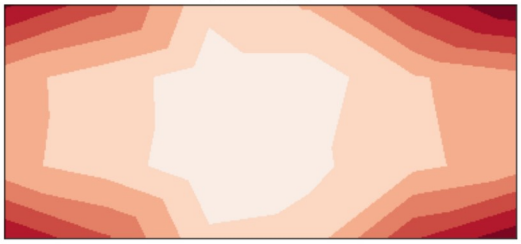


Mode 2 @ 13.543036 Hz

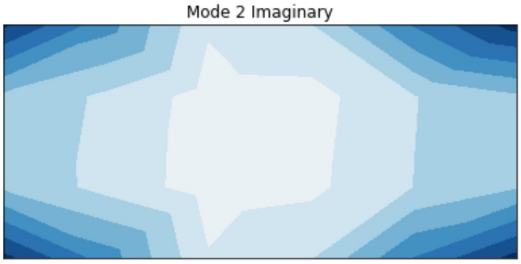


Mode 2 Phase

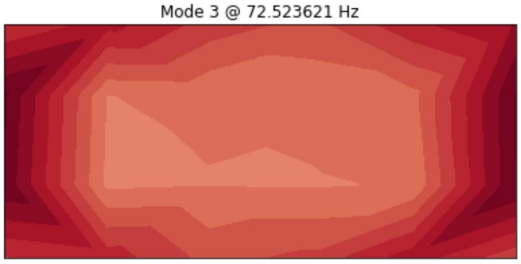
Mode 2 Real



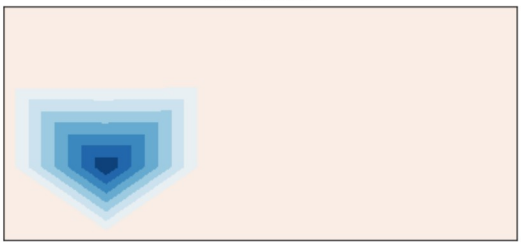
Mode 2 Imaginary



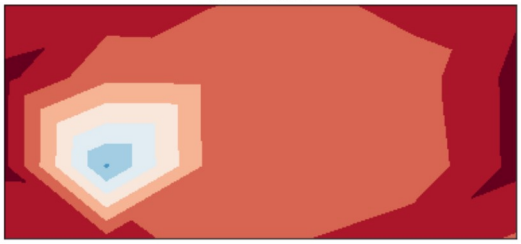
Mode 3 @ 72.523621 Hz



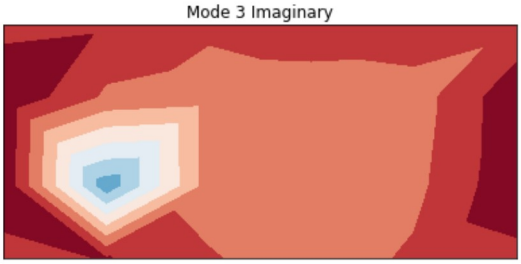
Mode 3 Phase

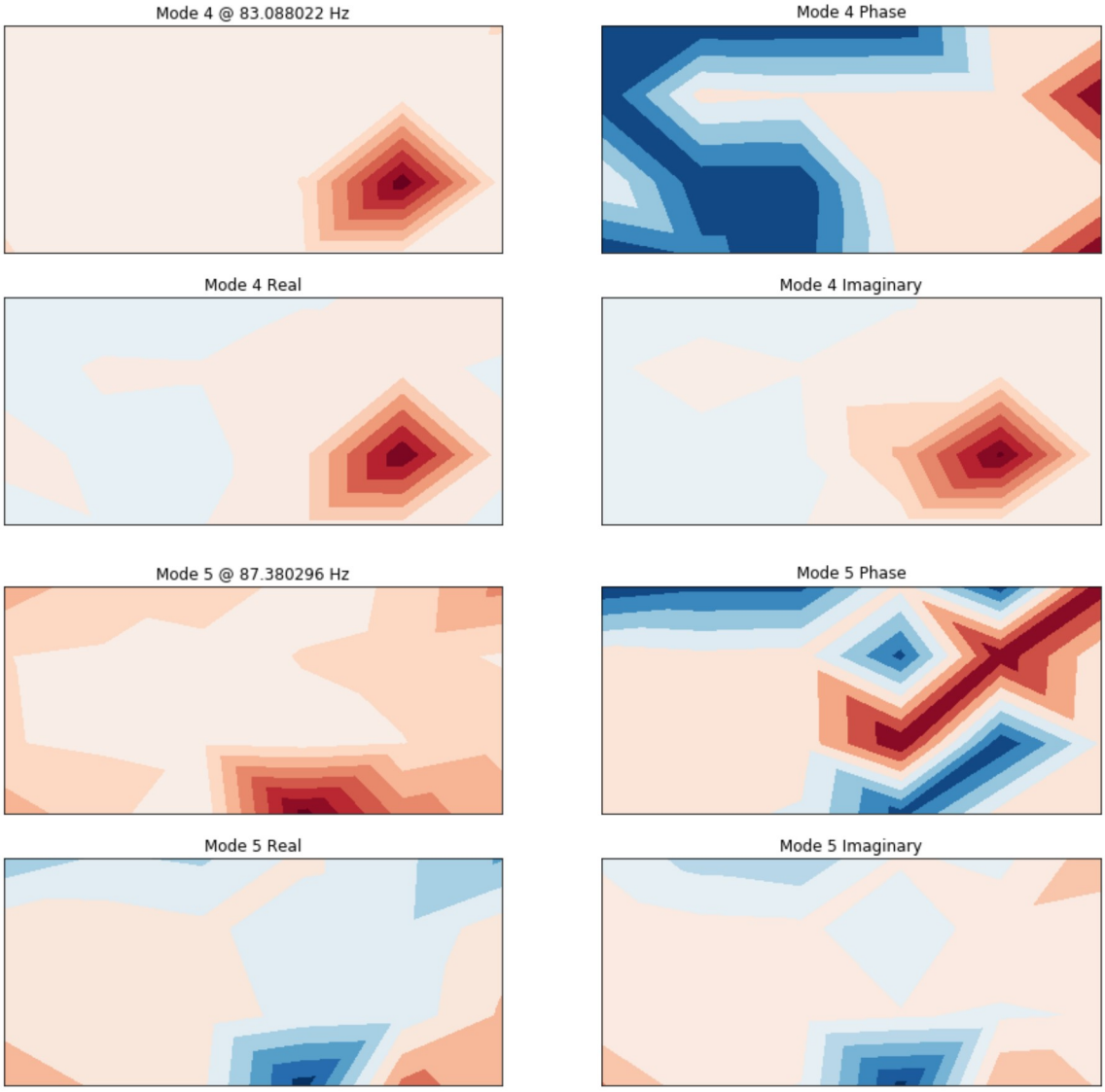


Mode 3 Real



Mode 3 Imaginary



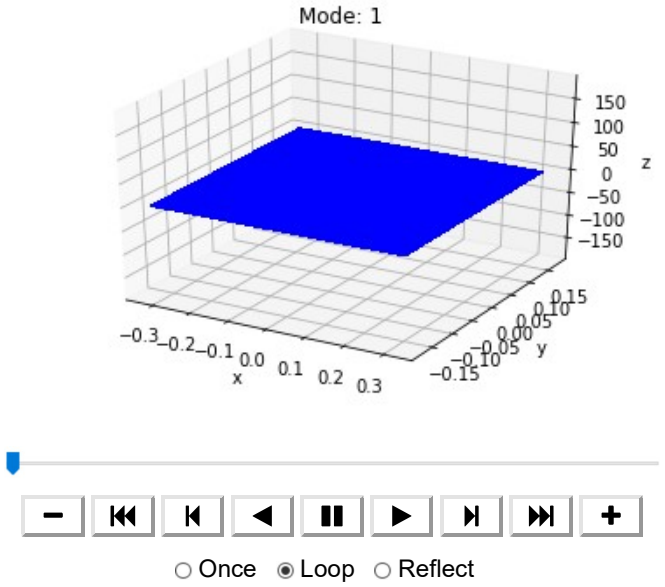


- animate selected mode shapes. What do you observe?

In [47]:
```
import matplotlib.animation as animation
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

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

    x = coordinates[:,1]
    y = coordinates[:,2]
    z = v

    x = np.reshape(x,(4,6))
    y = np.reshape(y,(4,6))
    z = np.reshape(z,(4,6))

    #------------------------------------------------------------
    # set up figure and animation
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    sc = ax.plot_surface(x,y,z)

    def animate(i):
        # scale factor for plotting
        s = np.sin(1/25*2*np.pi*i)
        Vs = z*s # sinusoidal scaled modeshape vector

        ax.clear()
        sc = ax.plot_surface(x,y,Vs,cmap=cm.brg,linewidth=0, antialiased=False)
        ax.set_xlabel('x')
        ax.set_ylabel('y')
        ax.set_zlabel('z')
        ax.set_zlim([-z.max(),z.max()])
        ax.set_title("Mode: %i" %(modenumber))

        return sc

    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=['-v
codec', 'libx264'])
    elif whatTOdo == 'justShow':
        plt.close()
        return ani
```

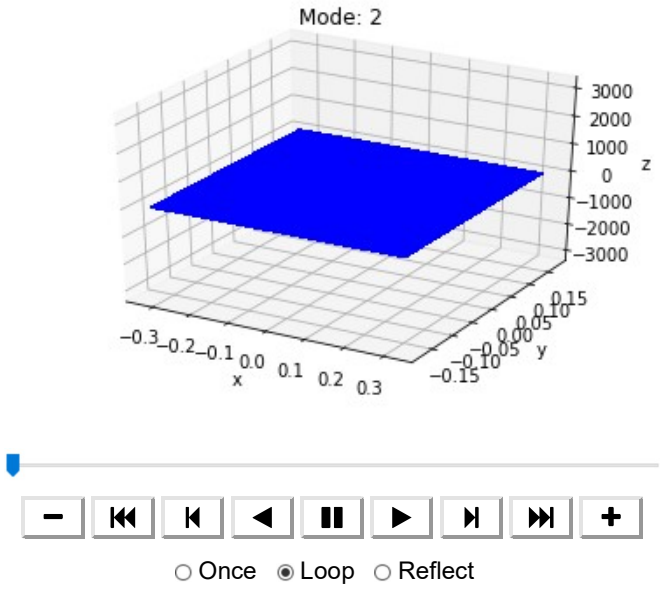
In [48]:
```
giveMeAnimation(np.abs(Mode_shapes_peakpicking[:,0]),coordinates,'justShow',1)
```

Out[48]:



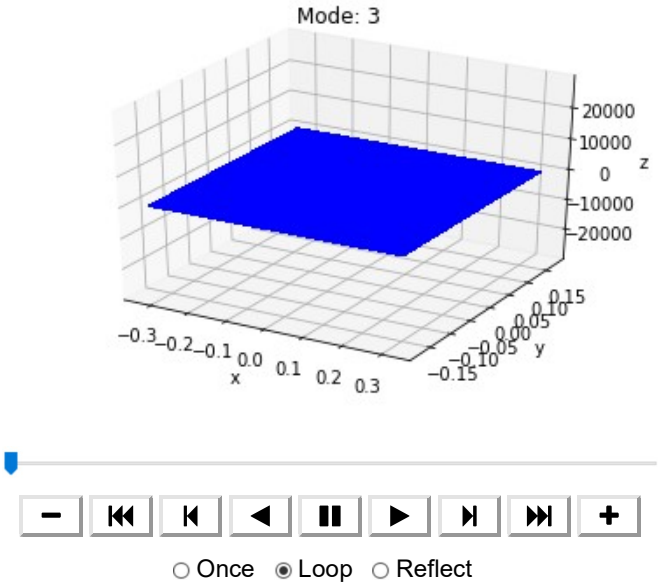
In [49]:
```
giveMeAnimation(np.abs(Mode_shapes_peakpicking[:,1]),coordinates,'justShow',2)
```

Out[49]:



In [50]:
```
giveMeAnimation(np.abs(Mode_shapes_peakpicking[:,2]),coordinates,'justShow',3)
```

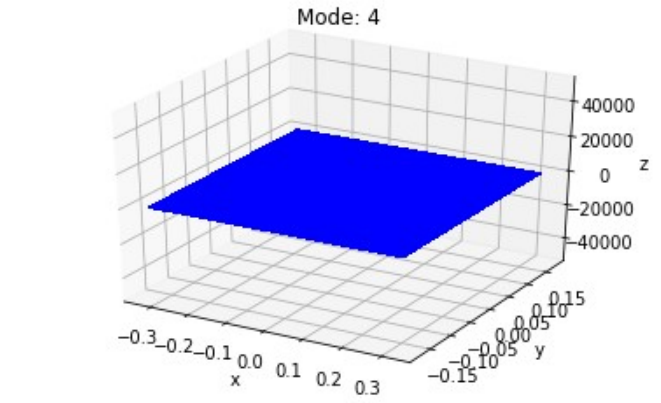
Out[50]:



○ Once ◉ Loop ○ Reflect

In [51]:
```
giveMeAnimation(np.abs(Mode_shapes_peakpicking[:,3]),coordinates,'justShow',4)
```

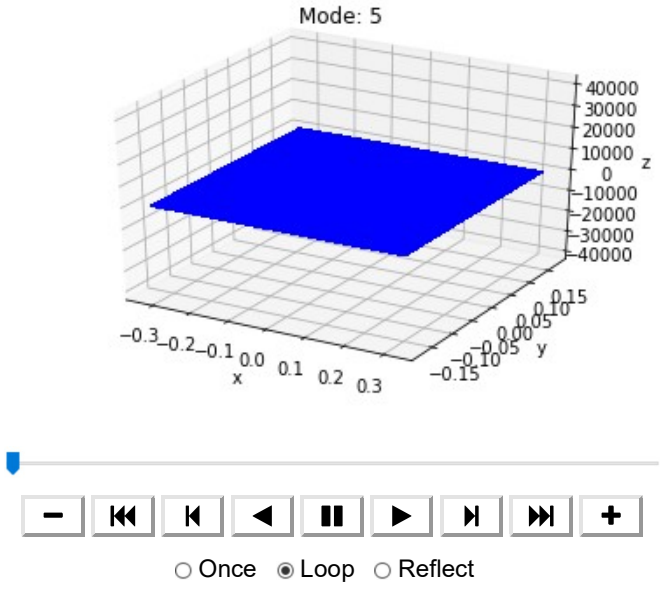
Out[51]:



○ Once ◉ Loop ○ Reflect

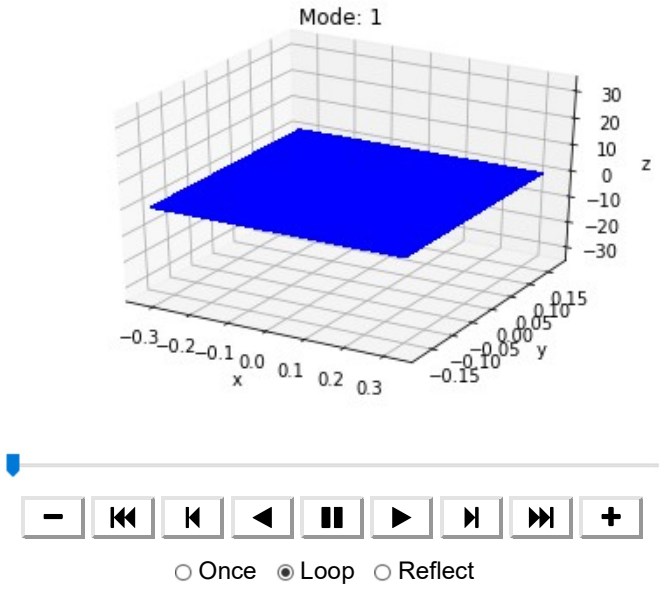
In [52]:
```
giveMeAnimation(np.abs(Mode_shapes_peakpicking[:,4]),coordinates,'justShow',5)
```

Out[52]:



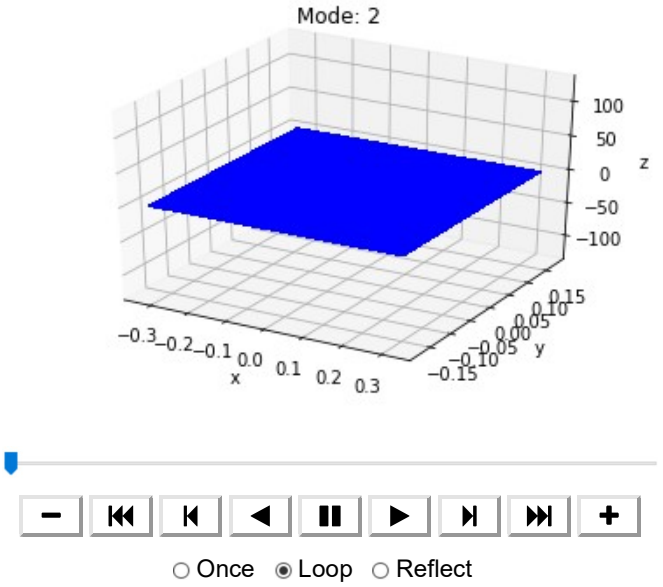
In [53]:
```
giveMeAnimation(np.abs(Mode_shapes_circle[:,0]),coordinates,'justShow',1)
```

Out[53]:



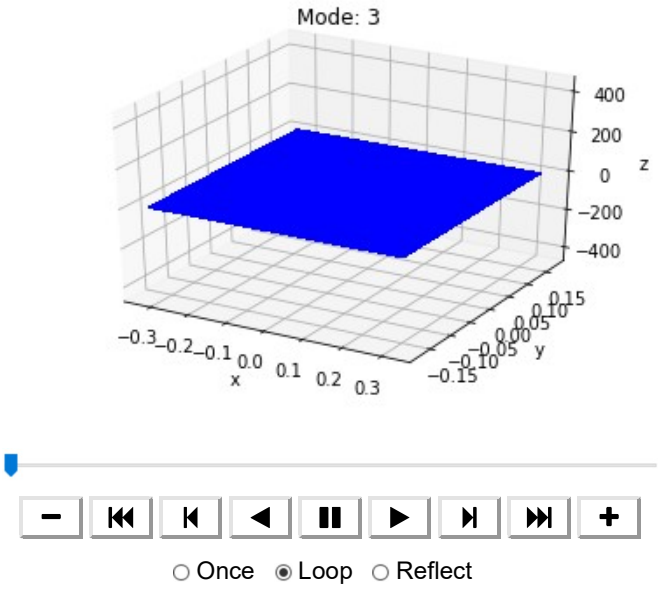
In [54]:
```
giveMeAnimation(np.abs(Mode_shapes_circle[:,1]),coordinates,'justShow',2)
```

Out[54]:



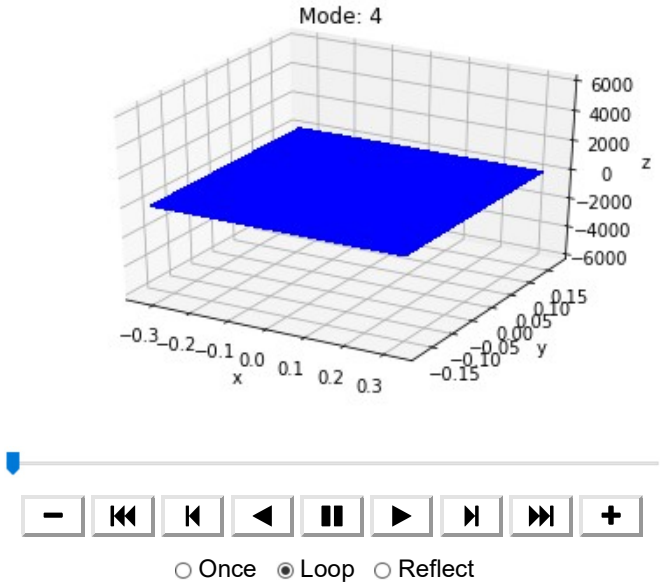
In [55]:
```
giveMeAnimation(np.abs(Mode_shapes_circle[:,2]),coordinates,'justShow',3)
```

Out[55]:



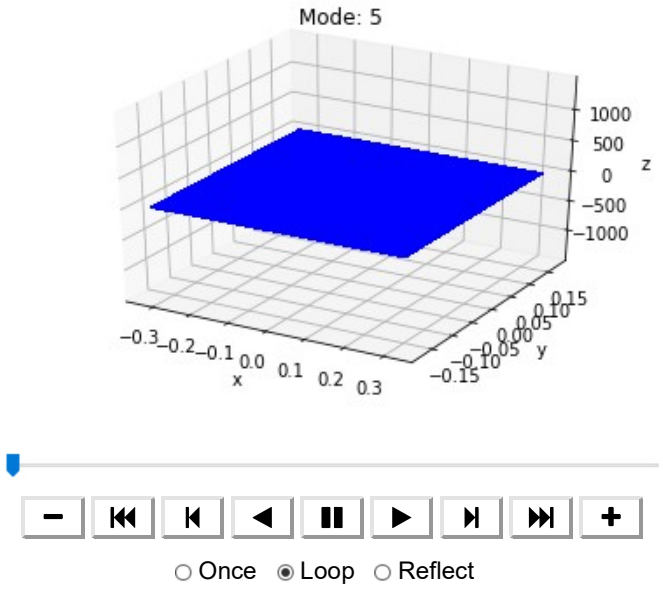
In [56]:
```
giveMeAnimation(np.abs(Mode_shapes_circle[:,3]),coordinates,'justShow',4)
```

Out[56]:



In [57]:
```
giveMeAnimation(np.abs(Mode_shapes_circle[:,4]),coordinates,'justShow',5)
```

Out[57]:



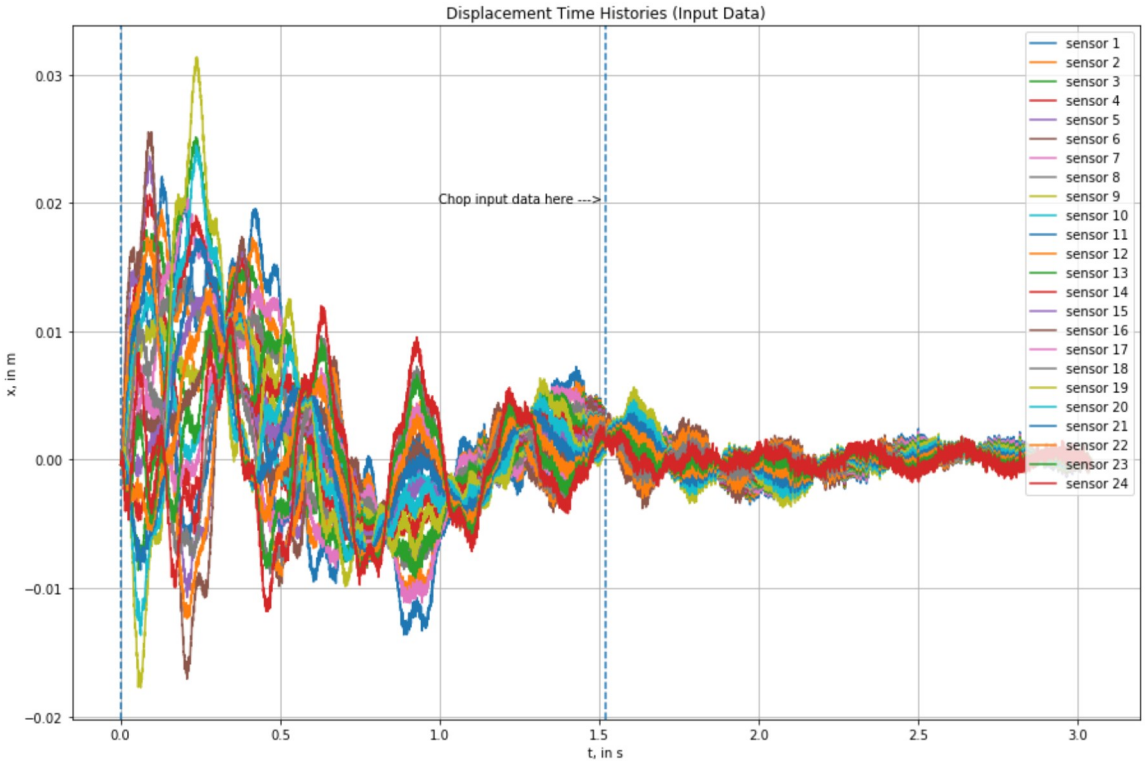
Ibrahim Time Domain Method

Implement the Ibrahim Time Domain method to use it on the data from the impact hammer experiemnt. Since the ITD operates on the free oscillation response in terms of the degrees of freedom (displacements in our case) you need to either compute the impulse respone from the accelerance using an inverse FFT or use the provided displacement data in `measurement-ITD.txt` .

In [58]:
```
def ITD(D,dt=1.0,n1=1,n3=17,tol=1.0):
    """Ibrahim time domain identification alogrithm

    Parameters
    ----------
    D : array(N,M)
        matrix containing data at N measurement positions
        time sampled at M times
    dt : float
        sampling interval
    n1 : int
        number samples to shift between measurement matrix X1 and
        its time shifted companion X2
    n3 : int
        number samples to shift between upper and lower part of
        X1 and X2, respectively
    tol : float
        tolerance for the MSCCF used to sort out computational modes.
        only modes abs(1-abs(MSCCF))<=tol are returned.

    Returns
    -------
    lam : array(K)
        identified eigenvalues, only positive frequencies are returned
    V : array(N,K)
        corresponding eigenvectors
    MSCCF : array(K)
        the mode shape correlation and confidence factor
    """

    from scipy.linalg import pinv

    # 1) determine which samples should be selected in X1 and X2
    Nt = D.shape[1] # Total number of time samples
    Ndof = D.shape[0] # Total number of (real) measurement locations

    It1 = np.arange(0,Nt-n3-n1,1)

    # 2) select samples from data matrix (without additional n2 shift)
    X1u = D[:,It1] # upper part
    X1l = D[:,It1+n3] # lower part (shifted by n3 samples)
    X2u = D[:,It1+n1] # time shifted companion (shifted by n1 samples)
    X2l = D[:,It1+n1+n3]

    # 3) assemble X1 and X2
    X = np.vstack((X1u, X1l))
    X_hat = np.vstack((X2u, X2l))
    #print("X.shape = ", X.shape) # DEBUG

    # 4) compute A: use a robust inverse like numpy's linalg.pinv, or solve line
ar systems
    A_1 = X_hat @ X.T @ pinv(X @ X.T)
    A_2 = X_hat @ X_hat.T @ pinv(X @ X_hat.T)
    A = (A_1 + A_2)/2

    # 5) solve eigenvalue problem
    S, V_comp = eig(A)
    #print(f"V_comp.shape = {V_comp.shape}") # DEBUG

    # 6) convert eigenvalues
    lam_comp = np.log(S)/(dt*n1)
    #print("lam_comp =", W_comp.shape, W_comp) # DEBUG
    #print("V_comp.shape", V_comp.shape) # DEBUG

    # 7) compute MSCCF and choose what to return
    V = np.empty((Ndof, 2*Ndof), dtype=complex)
    lam = np.empty(2*Ndof, dtype=complex)
    MSCCF = np.empty(2*Ndof, dtype=complex)
```

- use the ITD to identify modes from the impulse response. Choose what data to take (which time range is useful?), and what sampling parameters are suitable. Plot the input data over time (all signals in one plot).

In [59]:
```
# Load and assign data for ITD
data_ITD = loadtxt('measurement-ITD.txt').T

t = data_ITD[0,:]
D = data_ITD[1:min(data_ITD.shape),:]

dt = t[1] - t[0] # Sampling intervall; assuming constant dt

# Load sensor locations
sensor_locations = loadtxt('measurement-coordinates.txt')
sensor_locations_x = sensor_locations[:,1]
sensor_locations_y = sensor_locations[:,2]

# Chop input data
p_start = int(0 / dt)
p = max(data_ITD.shape) // 2 # Index
t_selected = t[p_start:p]
D_selected = D[:,p_start:p]

# Check displacements plot
plt.figure(figsize = [15,10])
plt.xlabel('t, in s')
plt.ylabel('x, in m')
plt.title('Displacement Time Histories (Input Data)')
plt.axvline(t_selected[-1], linestyle='--')
plt.axvline(t_selected[1], linestyle='--')
plt.annotate("Chop input data here --->", (t_selected[-1050], 0.02))
for i in np.arange(min(D.shape)) :
    plt.plot(t, D[i,:],label = 'sensor ' + str(i+1))

plt.legend(loc = 1)
plt.grid()
```

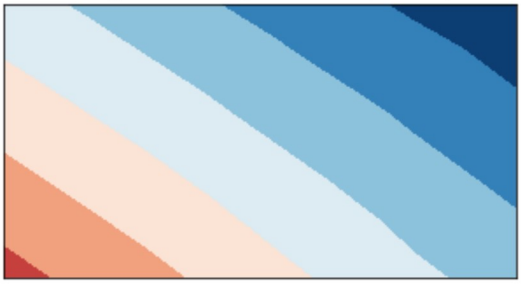


- plot the obtained modes and give their natural frequencies, damping factors, and MSCCFs.

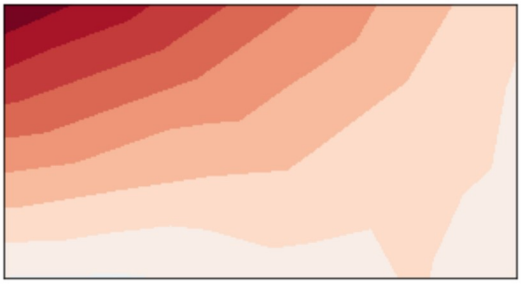
In [60]:
```
# Compute ITD
tol = 0.05
ITD_tol = tol
n1 = 1
n3 = 17
print(f"Expected frequency range: {0}-{1/(2*n1*dt):.2f}Hz")
lam, V_ITD, MSCCF = ITD(D, dt, n1, n3, tol=1000) # For demonstration: don't disc
ard computational modes.
#print(lam.shape, V.shape, MSCCF.shape, p) # DEBUG

# Plot all modes (also the computational ones)
for i in range(V_ITD.shape[1]):
    fig,ax = plotmode2d(np.real(V_ITD[:, i]), sensor_locations_x, sensor_locatio
ns_y)

    # Compute modal parameters from identified lambda
    r = np.real(lam[i])/np.imag(lam[i])
    zeta = np.sqrt(r**2/(1 + r**2)) # damping ratio
    omega_n = np.imag(lam[i])/np.sqrt(1-zeta**2)

    ax.set_title(f"MODE {i+1}: $f_d$ = {np.imag(lam[i])/2/np.pi:.2f}Hz, $\zeta$
= {zeta:.4f}, |MSSCF| = {np.abs(MSCCF[i]):.4f} $\Rrightarrow$ PHYSICAL MODE", lo
c='left')
    if abs(1-abs(MSCCF[i])) > tol:
        ax.set_title(f"MODE {i+1}: $f_d$ = {np.imag(lam[i])/2/np.pi:.2f}Hz, $\ze
ta$ = {zeta:.4f}, |MSSCF| = {np.abs(MSCCF[i]):.4f}  $\Rrightarrow$ COMPUTATIONAL
MODE", loc='left')
```

```
Expected frequency range: 0-1000.00Hz

G:\anaconda3\lib\site-packages\ipykernel_launcher.py:108: RuntimeWarning: More
than 20 figures have been opened. Figures created through the pyplot interface
(`matplotlib.pyplot.figure`) are retained until explicitly closed and may cons
ume too much memory. (To control this warning, see the rcParam `figure.max_ope
n_warning`).
```

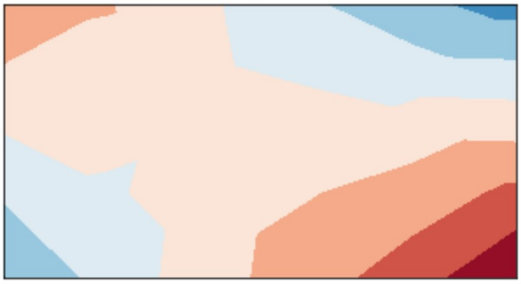
MODE 1: f_d = 3.35Hz, ζ = 0.0525, |MSSCF| = 0.9676 ⇒ PHYSICAL MODE



MODE 2: f_d = 8.43Hz, ζ = 0.1359, |MSSCF| = 1.2001 ⇒ COMPUTATIONAL MODE



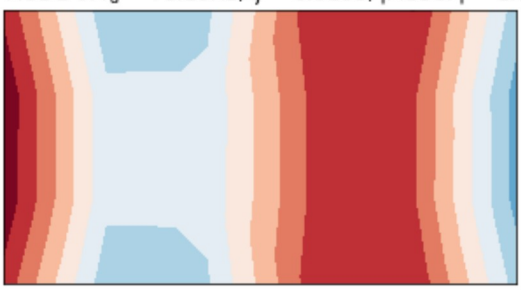
MODE 3: f_d = 16.62Hz, ζ = 0.1702, |MSSCF| = 1.1212 ⇒ COMPUTATIONAL MODE



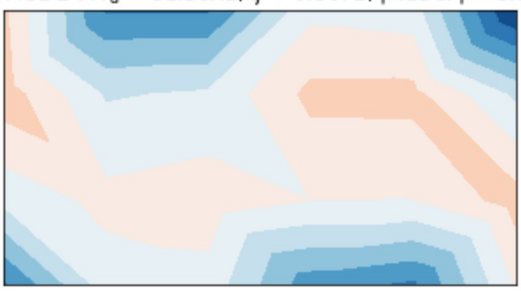
MODE 4: f_d = 16.87Hz, ζ = 0.1658, |MSSCF| = 1.8020 ⇒ COMPUTATIONAL MODE

MODE 5: f_d = 37.45Hz, ζ = 0.0051, |MSSCF| = 0.9959 ⇒ PHYSICAL MODE

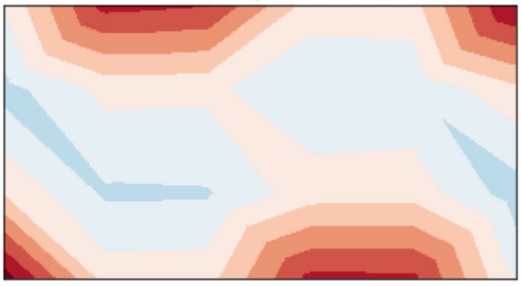
MODE 6: f_d = 73.19Hz, ζ = 0.0108, |MSSCF| = 1.0092 ⇒ PHYSICAL MODE



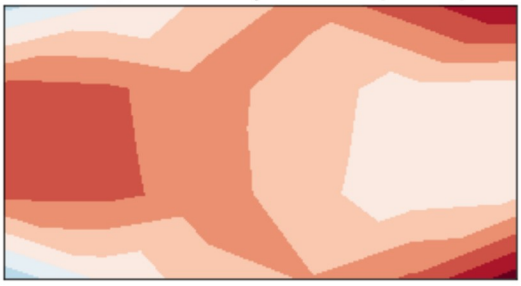
MODE 7: f_d = 81.30Hz, ζ = 0.0072, |MSSCF| = 0.8559 ⇒ COMPUTATIONAL MODE



MODE 8: f_d = 83.23Hz, ζ = 0.0028, |MSSCF| = 1.0544 ⇒ COMPUTATIONAL MODE



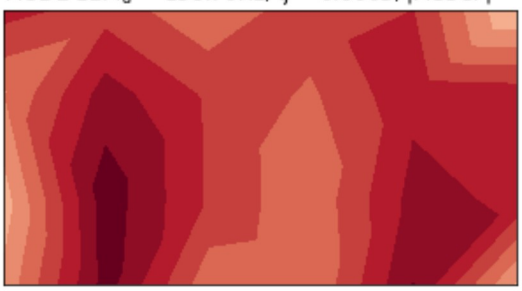
MODE 9: f_d = 86.74Hz, ζ = 0.0016, |MSSCF| = 0.7150 ⇒ COMPUTATIONAL MODE



MODE 10: f_d = 97.14Hz, ζ = 0.0080, |MSSCF| = 1.0611 ⇒ COMPUTATIONAL MODE

MODE 11: f_d = 109.24Hz, ζ = 0.0095, |MSSCF| = 0.5951 ⇒ COMPUTATIONAL MODE

MODE 12: f_d = 138.70Hz, ζ = 0.0003, |MSSCF| = 0.9513 ⇒ PHYSICAL MODE

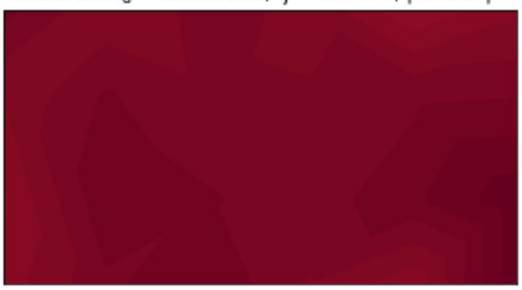


MODE 13: f_d = 144.36Hz, ζ = 0.2865, |MSSCF| = 25.6860 ⇒ COMPUTATIONAL MODE

MODE 14: f_d = 146.65Hz, ζ = 0.0027, |MSSCF| = 0.3496 ⇒ COMPUTATIONAL MODE

MODE 15: f_d = 183.91Hz, ζ = 0.0383, |MSSCF| = 0.6511 ⇒ COMPUTATIONAL MODE

MODE 16: f_d = 205.86Hz, ζ = 0.0476, |MSSCF| = 1.6644 ⇒ COMPUTATIONAL MODE



MODE 17: f_d = 215.44Hz, ζ = 0.0165, |MSSCF| = 0.5168 ⇒ COMPUTATIONAL MODE

MODE 18: f_d = 227.05Hz, ζ = 0.0102, |MSSCF| = 0.4005 ⇒ COMPUTATIONAL MODE

MODE 19: f_d = 243.23Hz, ζ = 0.0129, |MSSCF| = 0.3760 ⇒ COMPUTATIONAL MODE

MODE 20: f_d = 261.30Hz, ζ = 0.0065, |MSSCF| = 1.1003 ⇒ COMPUTATIONAL MODE

MODE 21: f_d = 276.38Hz, ζ = 0.0048, |MSSCF| = 1.8177 ⇒ COMPUTATIONAL MODE

Compare mode shapes

Use the MAC / MSF to compare your identified modes with the deformation modes of the free-free plate.

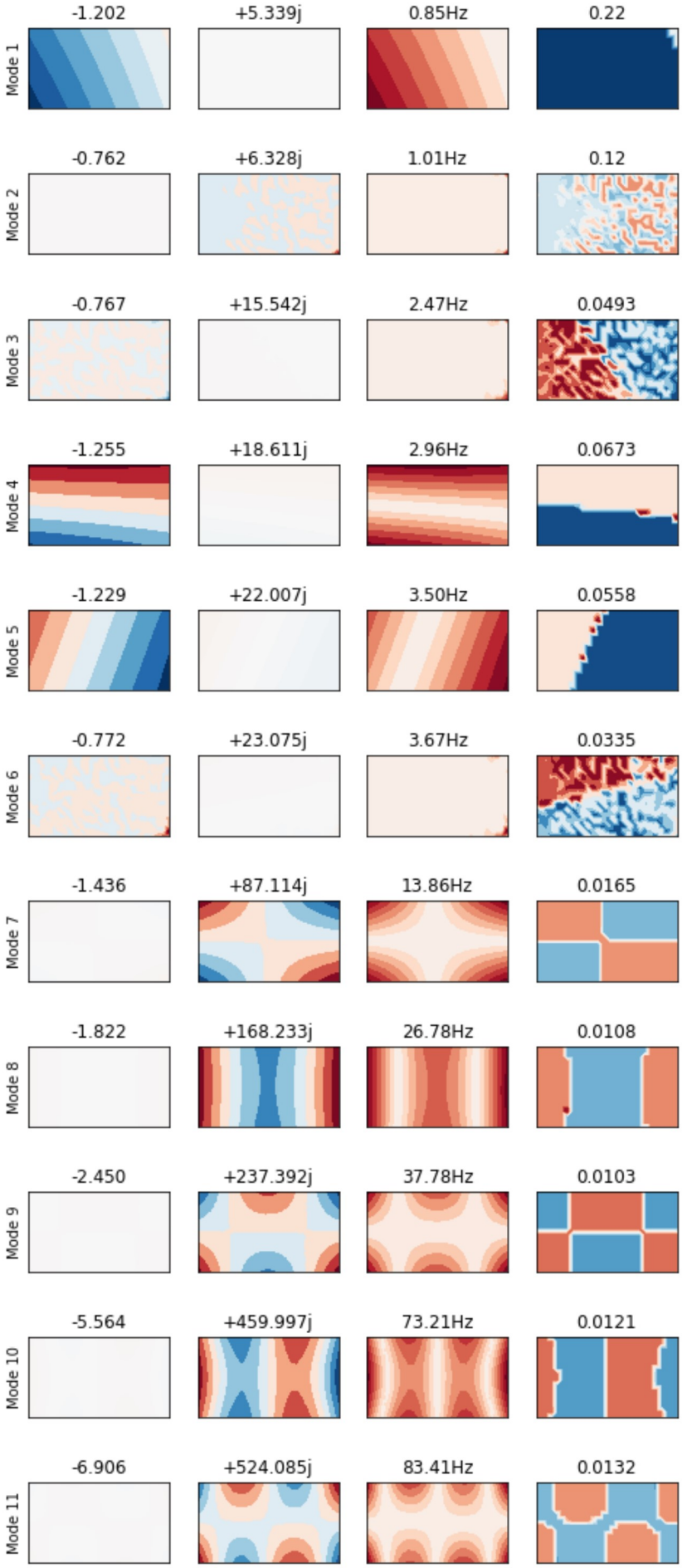
In [61]:
```
def mac_analysis(V_1, W_1, V_2, W_2, plot=False, plt_title="MAC analysis"):
    """Compute the MAC-matrix between the "reference/numeric" modes V_1 and the
"identified" modes V_2. Plot the matrix"""

    MAC = np.zeros((V_1.shape[1], V_2.shape[1]))

    for i in range(V_1.shape[1]):
       for j in range(V_2.shape[1]):
           MAC[i,j] = np.abs((np.conj(V_1[:,i]).T @ V_2[:,j]))**2 / ((np.conj(V_
1[:,i]).T @ V_1[:,i])*(np.conj(V_2[:,j]).T @ V_2[:,j]))

    if plot: # Display matrix
        fig, ax = plt.subplots(figsize=(8,8))
        cax = ax.matshow(MAC)
        ax.set_ylabel("REFERENCE MODES (V_1)")
        ax.set_xlabel("IDENTIFIED MODES (V_2)")
        ax.set_title(plt_title)
        fig.colorbar(cax)

    return MAC

def msf_analysis(V_1, W_1, V_2, W_2, plot=False, plt_title="MSF analysis"):
    """Compute the MSF-matrix between the "experimental" modes V_1 and the "nume
ric" modes V_2. Plot the matrix"""
    MSF = np.zeros((V_1.shape[1], V_2.shape[1]), dtype=complex)
    #print(f"MSF.shape {MSF.shape}") # DEBUG

    for i in range(V_1.shape[1]):
        for j in range(V_2.shape[1]):
            MSF[i,j] = (np.conj(V_1[:,i]).T @ V_2[:,j]) / (np.conj(V_1[:,i]).T @
V_1[:,i])

    #print(f"MSF.shape {MSF.shape}") # DEBUG

    if plot:
        # Display matrix
        fig, axs = plt.subplots(ncols=2, figsize=(14,6))
        cax1 = axs[0].matshow(np.abs(MSF))
        axs[0].set_ylabel(plt_title + "\nREFERENCE MODES (V_1)")
        axs[0].set_xlabel("IDENTIFIED MODES (V_2)")
        axs[0].set_title("|MSF|")
        cax2 = axs[1].matshow(np.angle(MSF))
        axs[1].set_ylabel("REFERENCE MODES (V_1)")
        axs[1].set_xlabel("IDENTIFIED MODES (V_2)")
        axs[1].set_title("angle(MSF)")
        #fig.colorbar(cax)

    return MSF

def compare_mode_shape(v_1, lam_1, v_2, lam_2, sensor_locations, drive_point_loc
ation, plt_title="Corr. Mode Pair"):
    """Plot the mode shapes of the correlated mode pair to check"""

    def zeta(lam):
        r = np.real(lam)/np.imag(lam)
        return(np.sqrt(r**2/(1 + r**2)))

    fig,axs = plt.subplots(ncols=2,figsize=[8,4])
    plotmode2d(np.abs(v_1), sensor_locations[:,0], sensor_locations[:,1], ax=axs
[0])
    #axs[0].annotate('$Excitation$'%(i+1),drive_point_location,size=12)
    plotmode2d(np.abs(v_2), sensor_locations[:,0], sensor_locations[:,1], ax=axs
[1])
    #axs[1].annotate('$Excitation$'%(i+1),drive_point_location,size=12)

    axs[0].set_title(f"$f_d=$ {np.imag(lam_1)/2/np.pi:.2f},$\zeta=$ {zeta(lam_
1):.5f}")
    axs[1].set_title(f"$f_d=$ {np.imag(lam_2)/2/np.pi:.2f},$\zeta=$ {zeta(lam_
```

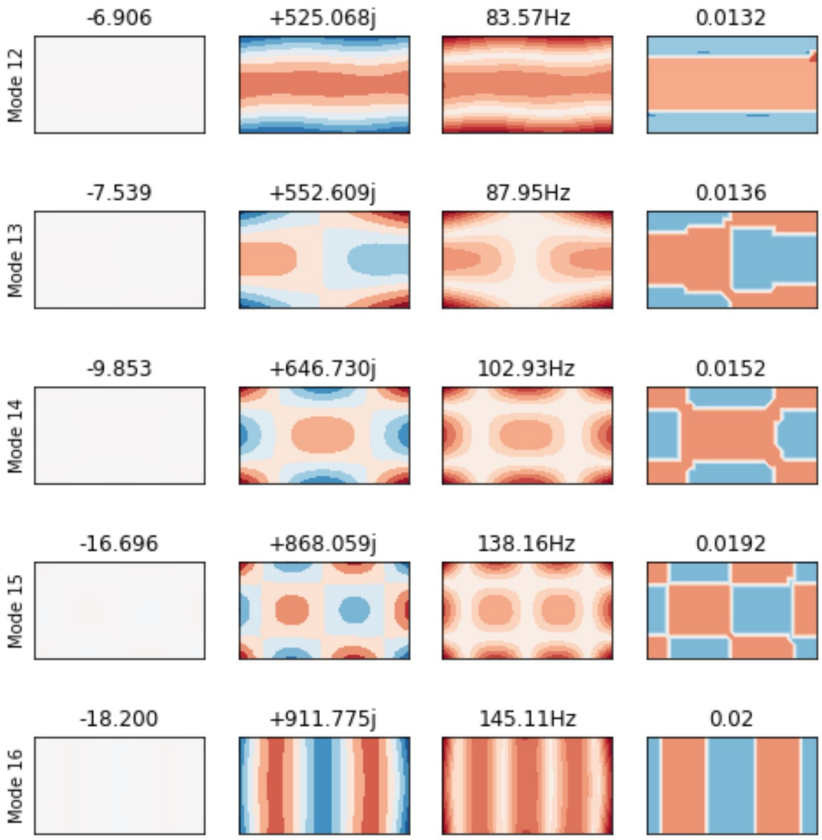
- Compute the free-free modes and interpolate/evaluate them at the measurement locations.

Numeric Reference Solution:

In [86]:
```
# load system matrices
M = sparse.csc_matrix(mmread('Ms.mtx'))#.toarray() # mass matrix
K = sparse.csc_matrix(mmread('Ks.mtx'))#.toarray() # stiffness matrix
X = mmread('X.mtx') # coodinate matrix with columns corresponding to x,y,z posit
ion of the nodes
C = sparse.csc_matrix(M.shape)
N = X.shape[0] # number of nodes

# for plotting
nprec = 6 # precision for finding uniqe values
# get grid vectors (the unique vectors of the x,y,z coodinate-grid)
xv = np.unique(np.round(X[:,0],decimals=nprec))
yv = np.unique(np.round(X[:,1],decimals=nprec))
zv = np.unique(np.round(X[:,2],decimals=nprec))

# point selection
tol = 1e-12

# select top of the plate
Nt = np.argwhere(np.abs(X[:,2]-X[:,2].max())<tol)[:,0]

# 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

# corner points
Nnoc = np.argwhere(np.all(np.vstack([np.abs(X[:,1]-X[:,1].max())<tol,np.abs(X[:,
0]-X[:,0].max())<tol,np.abs(X[:,2])<tol]),axis=0)).ravel()
Nnwc = np.argwhere(np.all(np.vstack([np.abs(X[:,1]-X[:,1].max())<tol,np.abs(X[:,
0]-X[:,0].min())<tol,np.abs(X[:,2])<tol]),axis=0)).ravel()
Nsoc = np.argwhere(np.all(np.vstack([np.abs(X[:,1]-X[:,1].min())<tol,np.abs(X[:,
0]-X[:,0].max())<tol,np.abs(X[:,2])<tol]),axis=0)).ravel()
Nswc = np.argwhere(np.all(np.vstack([np.abs(X[:,1]-X[:,1].min())<tol,np.abs(X[:,
0]-X[:,0].min())<tol,np.abs(X[:,2])<tol]),axis=0)).ravel()

Na = []
i = 0
for yi in yv[[1,5,-6,-2]]:
    for xi in xv[1::5]:
        i += 1
        Pi = [xi,yi,0]
        #print('P_%02i = %s'%(i,str(Pi)))
        Ni = np.argmin(np.sum((X-Pi)**2,axis=1))
        Na.append(Ni)
Na = np.array(Na)


E_s = 0.1e+9 # Young's modulus
r_s = 1e-3 # radius
l_s = 1 # length
k_s = r_s**2*pi*E_s/l_s

# Attach springs in x-direxction and test the plate upright
K_s = sparse.csc_matrix(K.shape)
for Ni in [Nnoc,Nsoc]: # the "top" corners
    # x-direction
    K_s[Ix[Ni],Ix[Ni]] += k_s
    # even softer springs in y and z-direction
    K_s[Iy[Ni],Iy[Ni]] += k_s/2
    K_s[Iz[Ni],Iz[Ni]] += k_s/2
# one spring at the bottom corner to remove last rigid body mode
K_s[Iz[Nnwc],Iz[Nnwc]] += k_s/10


# Set viscous damping
C_v = sparse.lil_matrix(M.shape)
```

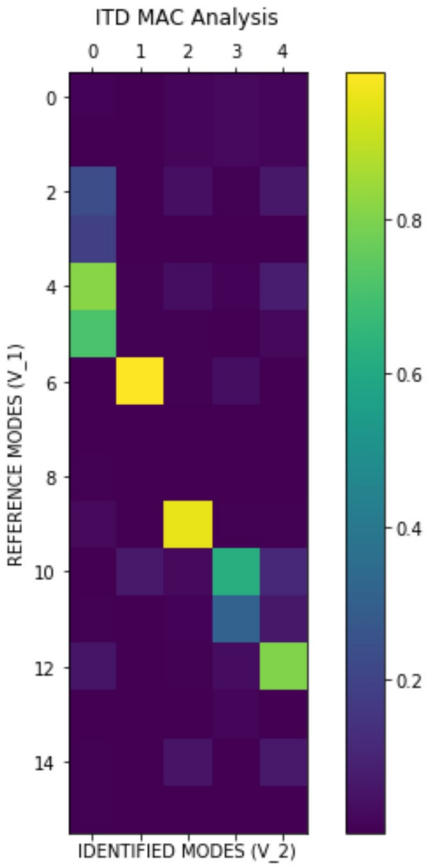
In [78]:
```
Ip = W1.imag >=0
for n,i in enumerate(np.argwhere(Ip).ravel()):
    fig,axs = plt.subplots(ncols=4,figsize=[8,1.5])
    lim = np.abs(V1[:3*N,i][Iz]).max()
    plotmode2d(V1[:3*N,i][Iz].real,*X[:,:2].T,ax=axs[0],lim=lim)
    plotmode2d(V1[:3*N,i][Iz].imag,*X[:,:2].T,ax=axs[1],lim=lim)
    plotmode2d(np.abs(V1[:3*N,i][Iz]),*X[:,:2].T,ax=axs[2],lim=lim)
    plotmode2d(np.angle(V1[:3*N,i][Iz]),*X[:,:2].T,ax=axs[3],lim=pi)
    r2 = (W1[i].real/W1[i].imag)**2

    axs[0].set_title('%+.3f'%W1[i].real)
    axs[1].set_title('%+.3fj'%W1[i].imag)
    axs[2].set_title('%.2fHz'%(W1[i].imag/2/pi))
    axs[3].set_title('%.3g'%(np.sqrt(r2/(1+r2))))

    axs[0].set_ylabel('Mode %i'%(n+1))
```

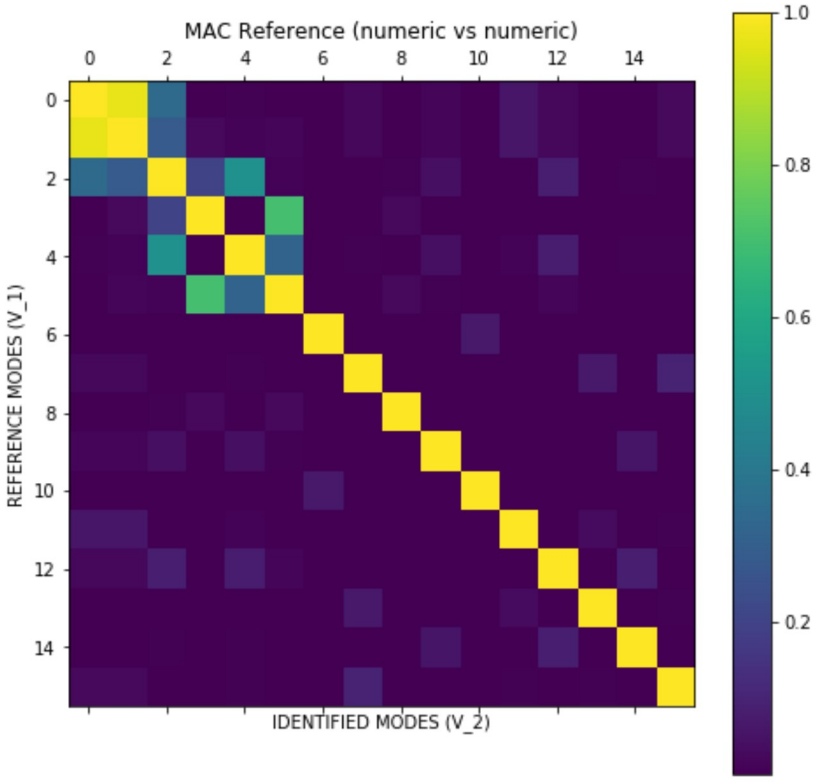

-1.202
+5.339j
0.85Hz
0.22
Mode 1
-0.762
+6.328j
1.01Hz
0.12
Mode 2
-0.767
+15.542j
2.47Hz
0.0493
Mode 3
-1.255
+18.611j
2.96Hz
0.0673
Mode 4
-1.229
+22.007j
3.50Hz
0.0558
Mode 5
-0.772
+23.075j
3.67Hz
0.0335
Mode 6
-1.436
+87.114j
13.86Hz
0.0165
Mode 7
-1.822
+168.233j
26.78Hz
0.0108
Mode 8
-2.450
+237.392j
37.78Hz
0.0103
Mode 9
-5.564
+459.997j
73.21Hz
0.0121
Mode 10
-6.906
+524.085j
83.41Hz
0.0132
Mode 11



- Compute the MAC-matrix between the "experimental" modes from peak-picking and the "numeric" free-free modes. Plot the matrix and ude it to identify the best-fitting "experimental" mode for each "numeric" deformation mode.

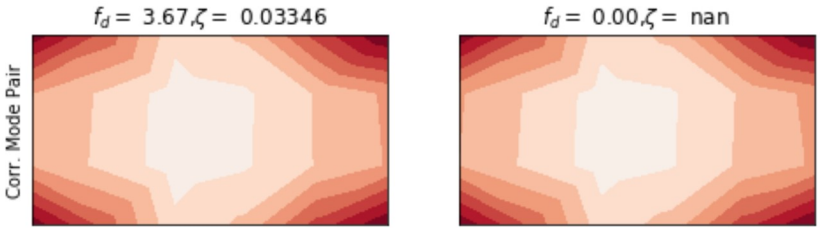
In [80]:
```
# MAC Analysis - PEAK-PICKING/REFERENCE
MAC_ITD = mac_analysis(V_z_ref_pos, W1_ref_pos, Mode_shapes_peakpicking, frequencies[peaks_idx_P05_rfft_window], plot=True, plt_title="ITD MAC Analysis")
MAC_reference =  mac_analysis(V_z_ref_pos, W1_ref_pos, V_z_ref_pos, W1_ref_pos, plot=True, plt_title="MAC Reference (numeric vs numeric)")
```





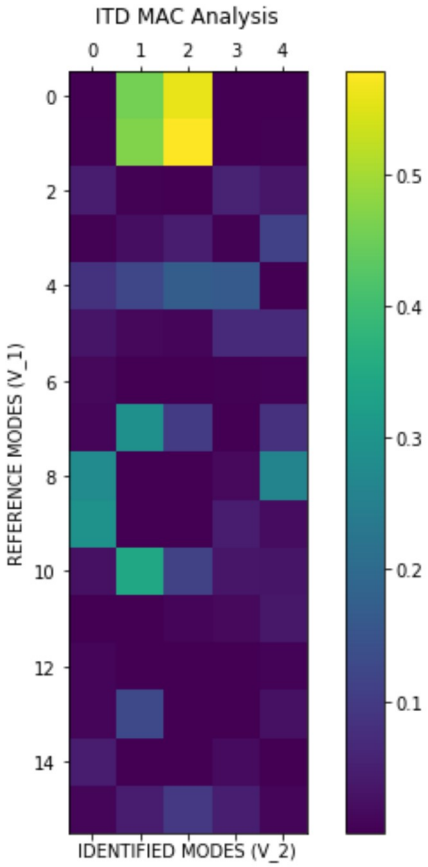
- Plot the mode shapes of the correlated mode pair to check. Do the frequencies match? Which threshold value of the MAC is suitable to select "correlated" modes?

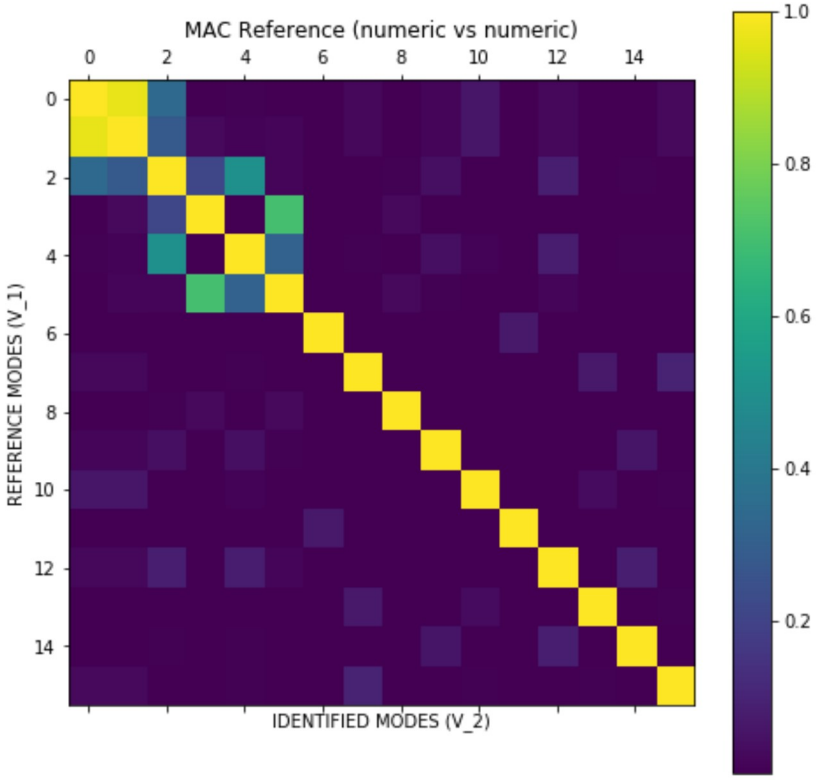
In [81]:
```
# Compare Modeshapes - PEAK-PICKING/REFERENCE
# e.g.: 7/2
ref_mode = 6
pp_mode = 1
compare_mode_shape(V_z_ref_pos[:,ref_mode], W1_ref_pos[ref_mode-1], Mode_shapes_
peakpicking[:,pp_mode], frequencies[peaks_idx_P05_rfft_window[pp_mode]], sensor_
locations[:,1:3], sensor_locations[4,1:2])
```



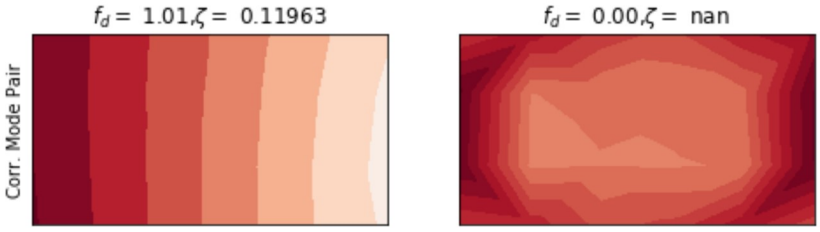
- Do the same for the modes from circle fitting. Mark the impact location in the plots. Give frequencies and damping ratios, as well as absolute value and phase of the MSF between "numeric" and "experimental" modes.

In [92]:
```
# MAC Analysis - CIRCLE/REFERENCE
wk_circle = np.mean(wk_all, axis=1)
MAC_ITD = mac_analysis(V_z_ref_pos, W1_ref_pos, Mode_shapes_circle, wk_circle/2/
np.pi, plot=True, plt_title="ITD MAC Analysis")
MAC_reference =  mac_analysis(V_z_ref_pos, W1_ref_pos, V_z_ref_pos, W1_ref_pos,
plot=True, plt_title="MAC Reference (numeric vs numeric)")
```

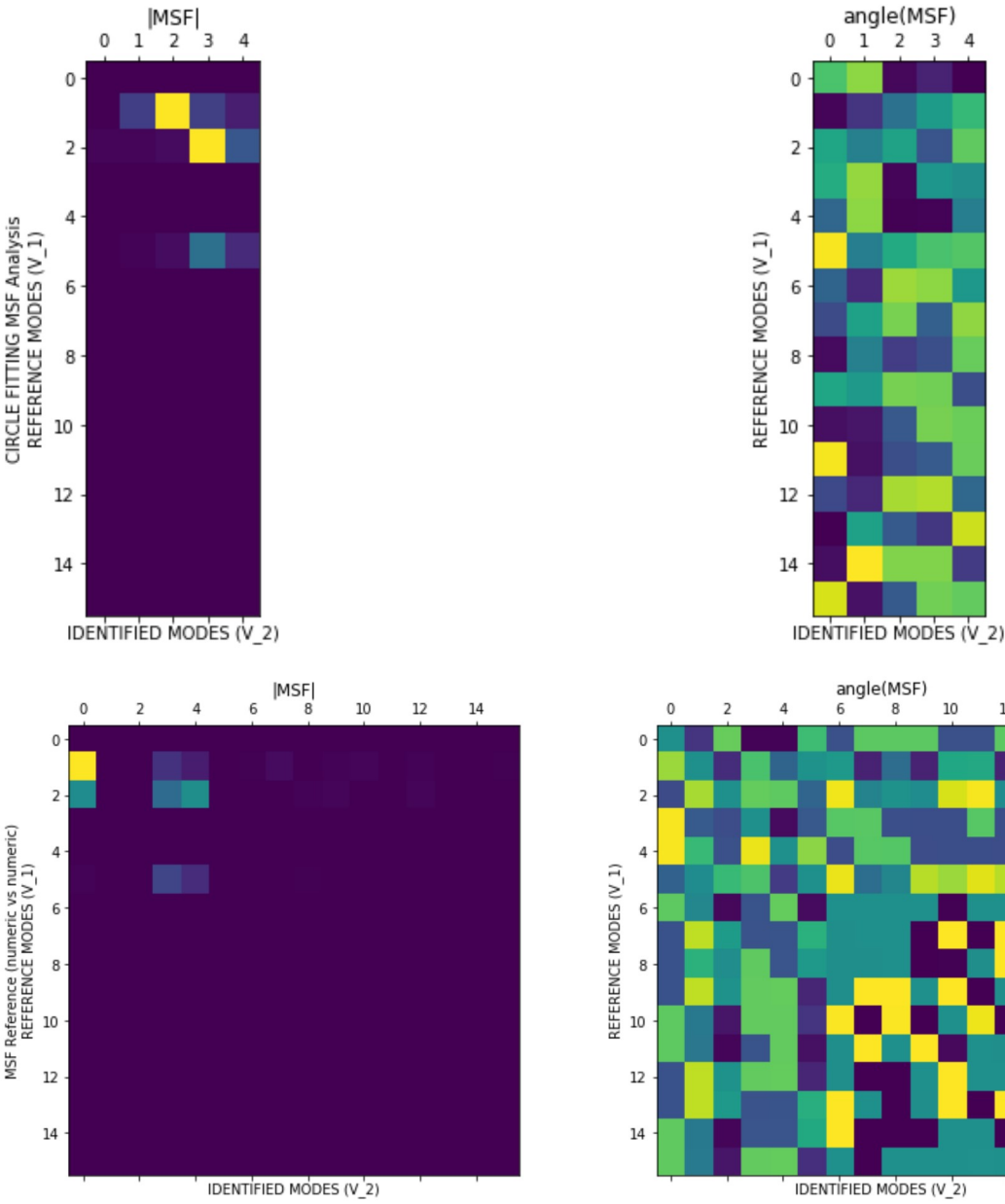




In [94]:
```
# Compare Modeshapes - CIRCLE/REFERENCE
ref_mode = 1
circle_mode = 2
compare_mode_shape(V_z_ref_pos[:,ref_mode], W1_ref_pos[ref_mode], Mode_shapes_ci
rcle[:,circle_mode], wk_circle[circle_mode], sensor_locations[:,1:3], sensor_loc
ations[4,1:2])
```



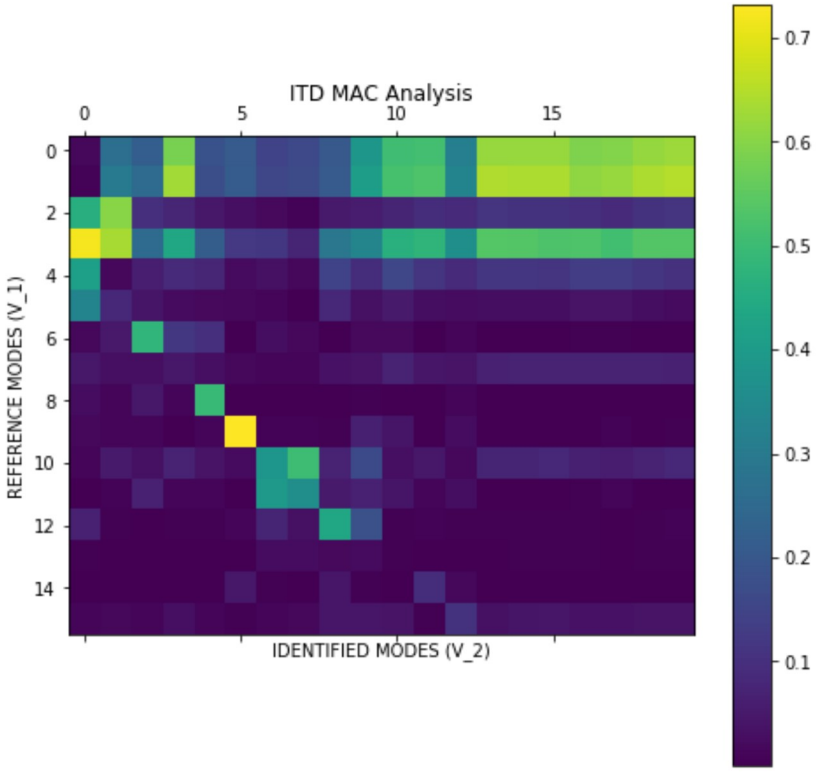
In [95]:
```
# MSF Analysis - ITD/REFERENCE
MSF_circle = msf_analysis(V_z_ref_pos, W1_ref_pos, Mode_shapes_circle, wk_all/2/
np.pi, plot=True, plt_title="CIRCLE FITTING MSF Analysis")
MSF_reference =  msf_analysis(V_z_ref_pos, W1_ref_pos, V_z_ref_pos, W1_ref_pos,
plot=True, plt_title="MSF Reference (numeric vs numeric)")
```

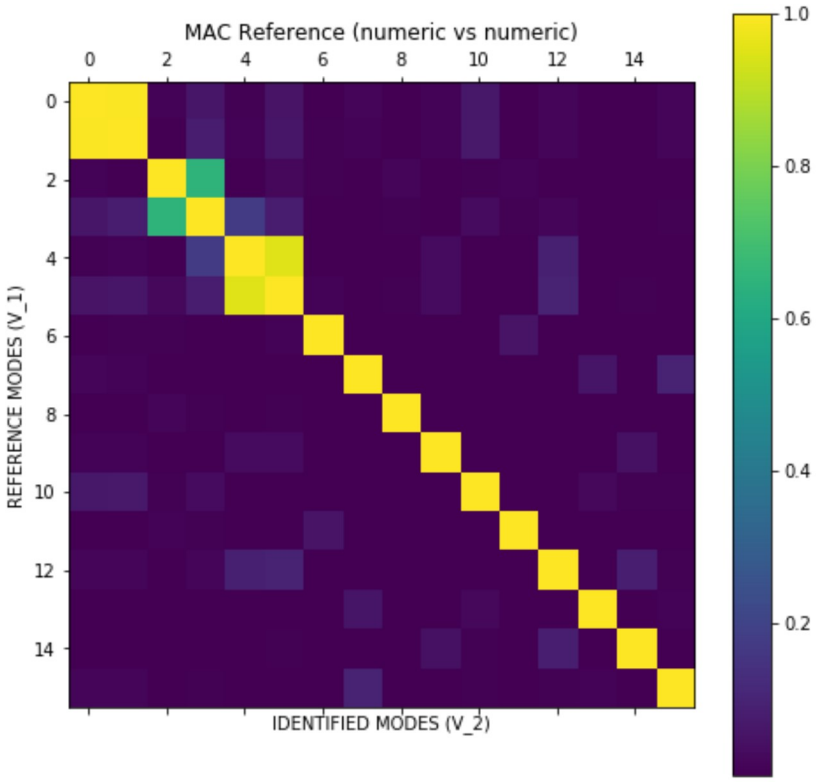


- Do the same for the modes form the ITD

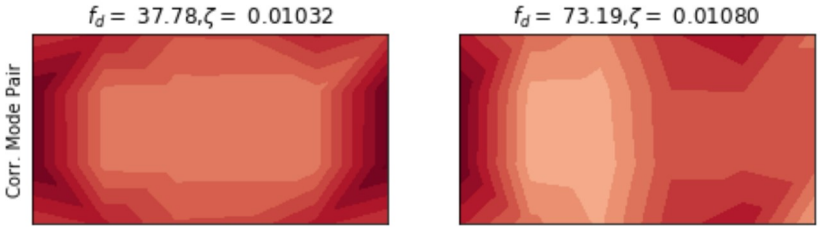
In [71]:
```
# Compute ITD
print(f"ITD-Parameters -> n1: {n1}, n3: {n3}, tol: {ITD_tol} \nExpected frequenc
y range: {0}-{1/(2*n1*dt):.2f}Hz")
lam, V_ITD, MSCCF = ITD(D, dt, n1, n3, tol=1)
print(f"--> {V_ITD.shape[1]} modes identified!")

# MAC Analysis - ITD/REFERENCE
MAC_ITD = mac_analysis(V_z_ref_pos, W1_ref_pos, V_ITD, np.imag(lam), plot=True,
plt_title="ITD MAC Analysis")
MAC_reference =  mac_analysis(V_z_ref_pos, W1_ref_pos, V_z_ref_pos, W1_ref_pos,
plot=True, plt_title="MAC Reference (numeric vs numeric)")
```

```
ITD-Parameters -> n1: 1, n3: 17, tol: 0.05
Expected frequency range: 0-1000.00Hz
--> 20 modes identified!

G:\anaconda3\lib\site-packages\ipykernel_launcher.py:8: ComplexWarning: Castin
g complex values to real discards the imaginary part
```





In [96]:
```
# Compare Modeshapes - ITD/REFERENCE
ref_mode = 9
itd_mode = 5
compare_mode_shape(V_z_ref_pos[:,ref_mode], W1_ref_pos[ref_mode-1], V_ITD[:,itd_
mode], lam[itd_mode], sensor_locations[:,1:3], sensor_locations[4,1:2])
```



In [73]:
```
# MSF Analysis - ITD/REFERENCE
MSF_ITD = msf_analysis(V_z_ref_pos, W1_ref_pos, V_ITD, np.imag(lam), plot=True,
plt_title="ITD MSF Analysis")
MSF_reference =  msf_analysis(V_z_ref_pos, W1_ref_pos, V_z_ref_pos, W1_ref_pos,
plot=True, plt_title="MSF Reference (numeric vs numeric)")
```