1 | Eigenvalue Problem

In [59]:

```
import numpy as np
from scipy.io import mminfo,mmread
import matplotlib.pyplot as plt
%matplotlib inline

from numpy.linalg import inv, eig
from numpy import sqrt, dot, sum, abs, diag, array

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

Eigenvalue Problems

In order to test different algorithms to compute eigenvalues we use a 4x4 matrix with eigenvalues 1, 2, 3, and 4. It is constructed as

$$A = SDS^{-1}$$

where D is a diagonal matrix containing the EVs.

In [60]:

```
np.random.seed(1)
D = np.array([1, 2, 3, 4]) # EIGENVALUES
S = (np.random.rand(len(D),len(D)) - 0.5)*2 # compute a random matrix with entries betw
een -1 and 1
A = np.dot(np.dot(S,np.diag(D)),inv(S)) # S*D*S^-1, computes a unitary similar matrix o
f D having the same EVs
D,S,A
```

Out[60]:

```
(array([1, 2, 3, 4]),
 array([[-0.16595599,  0.44064899, -0.99977125, -0.39533485],
        [-0.70648822, -0.81532281, -0.62747958, -0.30887855],
        [-0.20646505,  0.07763347, -0.16161097,  0.370439  ],
        [-0.5910955 ,  0.75623487, -0.94522481,  0.34093502]]),
 array([[ 4.00554954,  0.01354906,  1.19192526, -1.27636247],
        [ 2.77659659,  1.49989982,  3.16264938, -2.48173559],
        [ 1.45212322, -0.48865181,  6.86559345, -1.8724586 ],
        [ 4.12225091, -0.92237257,  9.49381671, -2.37104281]]))
```

Vector Interation

- also known as power method, power iteration, or von Mises iteration
- yields largest (in magnitude) eigenvalue and corresponding eigenvector
- converges badly if $\lambda_n/\lambda_{n-1} \approx 1$, i.e. the second largest EV is almost the same size as the largest

Use the recursion

$$b_{k+1} = \frac{Ab_k}{||Ab_k||}$$

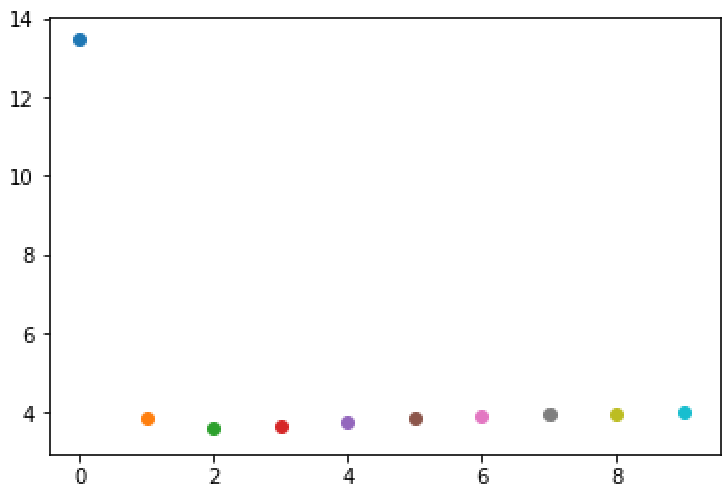
where b_k converges to the eigenvector and $||Ab_k||$ to the eigenvalue.

Hint: Depending on the eigenvalue the method might not converge to one fixed value. Try to implement a workaround in order to deal with the problem!

In [61]:

```
# eigenVal = vector_iter(A,10)

x = np.ones(len(A)) # one can start with any vector

for i in range(10) :
    # compute numerator
    num = np.dot(A,x)

    # compute denominator
    den = sum(abs(num)**2)**(1./2)
    #eigenVal = np.linalg.norm(num,2)

    # compute recursion
    x = num/den

    # plot the intermediate results (DEBUGGING)
    print('Current evaluation',i,'-->',den)
    plt.scatter(i, den)

print('\nLargest eigenvalue found:')
print(np.round_(den, decimals=1, out=None))

print('\nCorresponding eigenvecotor found:')
print(np.round_(x, decimals=1, out=None))
```

```
Current evaluation 0 --> 13.494287075225774
Current evaluation 1 --> 3.8923421594465695
Current evaluation 2 --> 3.614552826631208
Current evaluation 3 --> 3.654910499364184
Current evaluation 4 --> 3.762704545208437
Current evaluation 5 --> 3.862975312487751
Current evaluation 6 --> 3.9335469962764367
Current evaluation 7 --> 3.9756135798852528
Current evaluation 8 --> 3.9976324227962587
Current evaluation 9 --> 4.007617982928506

Largest eigenvalue found:
4.0

Corresponding eigenvecotor found:
[-0.5 -0.4  0.5  0.6]
```



Inverse Vector Iteration

If λ is an eigenvalue of A, then $1/\lambda$ is an eigenvalue of A^{-1}. Thus, we can do vector iteration on the inverse A^{-1} to find the Eigenvalue with the smallest magnitude. Thus, we have to compute the recursion

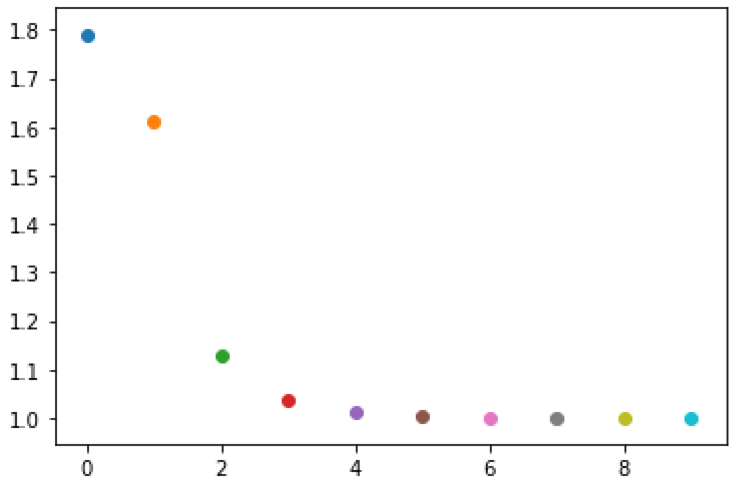
$$y_k = A^{-1}x_k$$

which can be done by pre-factorising $A = LU$.

In [62]:

```
# TODO: implement own LU pre-factorising
# eigenVal = vector_iter(inv(A),10)'

x = np.ones(len(A)) # one can start with any vector

for i in range(10) :
    # compute numerator
    num = np.dot(inv(A),x)

    # compute denominator
    den = sum(abs(num)**2)**(1./2)

    # compute recursion
    x = num/den

    # plot the intermediate results
    print('Current evaluation',i,'-->',1/den)
    plt.scatter(i, den)

print('\nLargest eigenvalue found:')
print(np.round_(1/den, decimals=1, out=None))

print('\nCorresponding eigenvecotor found:')
print(np.round_(x, decimals=1, out=None))
```

```
Current evaluation 0 --> 0.558365528631046
Current evaluation 1 --> 0.6207544440387353
Current evaluation 2 --> 0.8872586442640671
Current evaluation 3 --> 0.9632902755865219
Current evaluation 4 --> 0.9875404478461786
Current evaluation 5 --> 0.9956702927436427
Current evaluation 6 --> 0.9984664598730083
Current evaluation 7 --> 0.9994453940579879
Current evaluation 8 --> 0.9997942126045645
Current evaluation 9 --> 0.9999212025490571

Largest eigenvalue found:
1.0

Corresponding eigenvecotor found:
[-0.2 -0.7 -0.2 -0.6]
```



Inverse Vector Iteration with Shift

If λ is an eigenvalue of A then $\lambda - \sigma$ is an eigenvalue of $A - \sigma I$. The eigenvalue of the inverse $(A - \sigma I)^{-1} = B$ will be $\mu = \frac{1}{\lambda - \sigma}$. Thus, if $\lambda \approx \sigma$, vector iteration with B will yield the smallest (in magnitude) EV of A.

The iteration rule is then

$$b_{k+1} = (A - \sigma I)^{-1} b_k \text{ or } (A - \sigma I) b_{k+1} = b_k$$

The series converges to the same eigenvectors, the eigenvalues μ are related to the original ones λ via

$$\lambda = \sigma + \frac{1}{\mu}$$

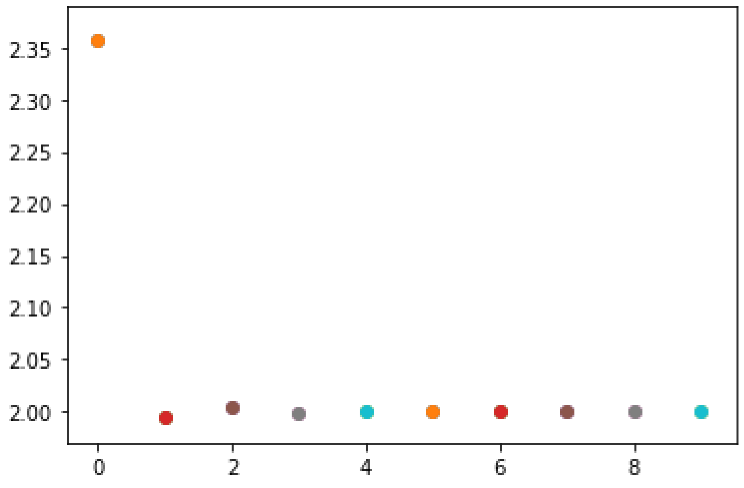
- σ is called the shift point
- a linear system has to be solved in each step
- for a constant shift point the solution of the linear system corresponds to a matrix multiplication

> If you shift to exactly one eigenvalue $A - \sigma I$ becomes ill-conditioned!

In [63]:

```
# choose shift point
sig = 1.7
B = inv(A-sig*np.diag(np.ones_like(D)))

#y = np.dot(Binv,np.ones_like(D)) # start value
x = np.ones(len(D))
for i in range(10) :
    # compute update
    num = np.dot(B,x)

    # compute denominator
    den = sum(abs(num)**2)**(1./2)

    # compute recursion
    x = num/den

    # Check sign
    eigV = np.dot(np.transpose(x),np.dot(B,x))/np.dot(np.transpose(x),x)

    # update lambda
    lamb = sig + 1/eigV

    # do fancy plot
    plt.scatter(i, lamb)

    # display some stuff & plot fancy
    print('Current evaluation',i,'-->',lamb)
    plt.scatter(i, lamb, marker='o')


print('\nEigenvalue found:')
print(np.round(sig+1/eigV, 2, out=None))

print('\nCorresponding eigenvecotor found:')
print(np.round_(x, decimals=2, out=None))
```

```
Current evaluation 0 --> 2.3565113390406536
Current evaluation 1 --> 1.9949654524961131
Current evaluation 2 --> 2.003932404040147
Current evaluation 3 --> 1.999436794382986
Current evaluation 4 --> 2.0002598041227637
Current evaluation 5 --> 1.9998888391860403
Current evaluation 6 --> 2.0000431417548095
Current evaluation 7 --> 1.9999800088866169
Current evaluation 8 --> 2.0000080976247667
Current evaluation 9 --> 1.999996401973084

Eigenvalue found:
2.0

Corresponding eigenvecotor found:
[-0.37  0.68 -0.06 -0.63]
```



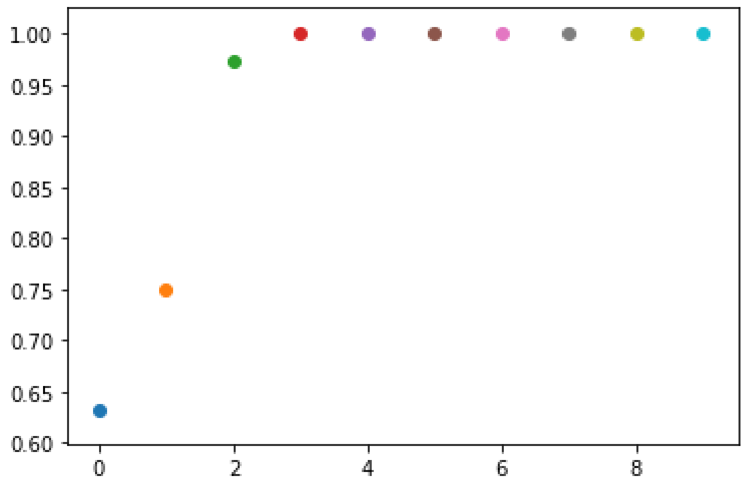
Rayleigh Quotient Iteration

Do (inverse) vector iteration but, adapt the shift point in every iteration.

In [64]:

```
#y = np.dot(Binv,np.ones_like(D)) # start value
sig = 0
B = inv(A-sig*np.diag(np.ones_like(D)))
x = np.ones(len(D))

for i in range(10) :

    # compute update
    num = np.dot(B,x)

    # compute denominator
    den = sum(abs(num)**2)**(1./2)

    # compute recursion
    x = num/den

    # Check sign
    eigV = np.dot(np.transpose(x),np.dot(B,x))/np.dot(np.transpose(x),x)

    # update lambda
    lamb = sig + 1/eigV

    # update shift point
    sig=np.dot(np.transpose(x),np.dot(A,x))/np.dot(np.transpose(x),x)
    B = inv(A-sig*np.diag(np.ones_like(D)))

    # display some stuff & plot fancy
    print('Current evaluation',i,'-->',lamb)
    plt.scatter(i, lamb, marker='o')

print('\nEigenvalue found:')
print(np.round(lamb, 2, out=None))

print('\nCorresponding eigenvector found:')
print(np.round_(x, decimals=2, out=None))
```

```
Current evaluation 0 --> 0.6322996987908873
Current evaluation 1 --> 0.7494067382127579
Current evaluation 2 --> 0.9736785316403405
Current evaluation 3 --> 0.9996039023892067
Current evaluation 4 --> 0.9999999084580931
Current evaluation 5 --> 0.9999999999999968
Current evaluation 6 --> 1.0000000000000002
Current evaluation 7 --> 1.0000000000000002
Current evaluation 8 --> 1.000000000000001
Current evaluation 9 --> 1.0000000000000002

Eigenvalue found:
1.0

Corresponding eigenvector found:
[0.17 0.74 0.22 0.62]
```



QR Algorithm

One can ompute all eigenvalues at once using the QR-Algorithm: Compute the QR decomposition of A, i.e.

$$A_k = Q_k R_k$$

and then apply the iteration rule

$$A_{k+1} = R_k Q_k$$

which converges to upper diagonal form with the eigenvalues of A in the diagonal.

> Can you compute the eigenvectors once the eigenvalues are known?

Yes!

1. Through shifted vector iteration, using the now known eigenvalues.
2. Or with

$$Q_\infty = \Pi_{k=1} Q_k$$

In [65]:

```
def eigQR(A, maxIter = 20):
    """ Returns eigenvalues of matrix A using the QR decomposition as a list [lambda1,
 ..., lambdaN].
        Also returns normalized eigenvectors as column vectors aranged into an array [v
1, ..., vN]

    """
    U = A

    # compute eigenvalues and eigenvectors
    eigenVecs = np.eye(U.shape[0]) # initialize identity matrix

    for k in range(maxIter):

        Q,R = np.linalg.qr(U) # QR decomposition of matrix A
        U = R @ Q # @ operator for matrix multiplication
        eigenVecs = eigenVecs @ Q # compute eigenvector matrix Q = I*Q_0*Q_1*...

    eigenVals = np.diag(U)
    return eigenVals, eigenVecs
```

In [66]:

```
# test our solver
eigenVals, eigenVecs = eigQR(A)
sigDigits = 2
print("Our QR-algorithm solver:")
print("Eigenvalues:", np.round(eigenVals, sigDigits))
print("Eigenvectors:\n", np.round(eigenVecs, sigDigits))
print("\n-----------------------------------------\n")
# numpy reference solver
w,v = np.linalg.eig(A)
print("Numpy reference solver:")
print("Eigenvalues:", np.round(w, sigDigits))
print("Eigenvectors:\n", np.round(v, sigDigits))
```

```
Our QR-algorithm solver:
Eigenvalues: [4. 3. 2. 1.]
Eigenvectors:
 [[-0.55  0.57 -0.45 -0.42]
 [-0.43  0.34  0.82  0.14]
 [ 0.53  0.2   0.32 -0.76]
 [ 0.49  0.72 -0.13  0.48]]

-----------------------------------------

Numpy reference solver:
Eigenvalues: [1. 4. 3. 2.]
Eigenvectors:
 [[ 0.17 -0.56 -0.66  0.37]
 [ 0.74 -0.43 -0.41 -0.68]
 [ 0.22  0.52 -0.11  0.06]
 [ 0.62  0.48 -0.62  0.63]]
```

In [67]:

```
for v, mu in zip(eigenVecs.T, eigenVals): # iterate over eigenvectors and matching eige
nvalues
    print("Eigenpair:", np.round(mu, sigDigits), ",", np.round(v, sigDigits))
    print("Av - lambda*v = ", np.round(A @ v - mu * v, 1)) # check if the eigenpairs so
lve the eigenproblem and show us
    print("-----------------------------------------\n")
```

```
Eigenpair: 4.0 , [-0.55 -0.43  0.53  0.49]
Av - lambda*v =  [-0. -0. -0. -0.]
-----------------------------------------

Eigenpair: 3.0 , [0.57 0.34 0.2  0.72]
Av - lambda*v =  [-0.1 -0.1  0.1  0.1]
-----------------------------------------

Eigenpair: 2.0 , [-0.45  0.82  0.32 -0.13]
Av - lambda*v =  [-0.3 -0.3  0.7  1. ]
-----------------------------------------

Eigenpair: 1.0 , [-0.42  0.14 -0.76  0.48]
Av - lambda*v =  [ -2.8  -4.7  -6.  -10.7]
-----------------------------------------
```

Subspace Iteration

Compute an orthonormal basis $X \in \mathbb{C}^{n\times p}$ of p vectors and use it in the iteration

$$Z_{k+1} = AX_k$$

where

$$X_k R_k = Z_k$$

is the QR factorization of Z_k.

The largest (in magnitude) eigenvalues appear in the diagonal of R_k.

In [68]:

```
p = 3 # subspace size -> compute p eigenvalues

# random starting basis
X = np.random.rand(len(D),p)
print("Random start matrix X of size", len(D),"x",  p,":\n", X)
```

```
Random start matrix X of size 4 x 3 :
 [[0.4173048  0.55868983 0.14038694]
 [0.19810149 0.80074457 0.96826158]
 [0.31342418 0.69232262 0.87638915]
 [0.89460666 0.08504421 0.03905478]]
```

In [69]:

```
def grBasis(V):
    '''
        Applies the Gram-Schmid process to a regular n x p matrix V and returns orthono
rmalized matrix U
    '''
    # init matrix U
    U = V
    # loop over all column matrices == basis vectors
    for k in range(V.shape[-1]):

        #print("k =", k) # debugging

        # sum up all projections
        sumOfProjections = np.zeros_like(V[:,k]) # initialize with 0 vector

        for j in range(k):
           #print("j =", j) # debugging
            sumOfProjections += np.vdot(V[:,k],U[:,j]) / np.vdot(U[:,j],U[:,j]) * U[:,j
]# projection of v_k onto u_j

        #print("sumOfProjections =", sumOfProjections) #debugging

        U[:,k] = V[:,k] - sumOfProjections # orthogonalize
        U[:,k] = U[:,k] / np.linalg.norm(U[:,k]) # normalize

    return U
```

In [70]:

```
U = grBasis(X)
print("Computed matrix U of size", len(D),"x",  p,":\n", U)
print("Orthogonal/unitary? U^H*U:\n", np.round(U.T.conj() @ U))
```

```
Computed matrix U of size 4 x 3 :
 [[ 0.39574193  0.29898247 -0.86547717]
 [ 0.18786523  0.67268214  0.26422952]
 [ 0.297229    0.49483316  0.3661662 ]
 [ 0.84838076 -0.46178823  0.21692012]]
Orthogonal/unitary? U^H*U:
 [[ 1. -0.  0.]
 [-0.  1. -0.]
 [ 0. -0.  1.]]
```

In [71]:

```
# perform subspace iteration

U = grBasis(X) # assign random orthonormal subspace basis matrix U = [u_1, ..., u_p] vi
a the Gram-Schmidt algorithm


for i in range(10):
    Z = A @ U # parallel power iteration in the n x p subspace, note the similarity to
 the power iteration!
    U, R = np.linalg.qr(Z) # qr factorization based on the classical Gram-Schmidt algor
ithm is numerically unstable -> np.linalg.qr() is used instead

print("The p =", R.shape[0], "largest eigenvalues are:", np.round(np.diag(R)))
```

```
The p = 3 largest eigenvalues are: [4. 3. 2.]
```

Submission Task 1: Function for Vector Iteration

Write a function for the vector iteration as well as the iteration for higher eigenvalues using the call-signature given below.

The function `vector_iteration` takes a matrix A as an input. The function should iterate until the relative norm of the update is smaller than a defined tolerance ϵ, i.e. until

$$\frac{||\lambda_{k+1} - \lambda_k||}{||\lambda_k||} < \epsilon$$

In case of non-convergence after a maximum number of iterations, the function should terminate with a warning. The function should return the eigenvalue, the corresponding eigenvector as well as the number of iterations. Use the call signature give below, which already contains useful default values for tolerance and maximum number of iterations.

In order to test your function apply use the four matrices

$$M_1 = \begin{bmatrix} 4.00554954 & 0.01354906 & 1.19192526 & -1.27636247 \\ 2.77659659 & 1.49989982 & 3.16264938 & -2.48173559 \\ 1.45212322 & -0.48865181 & 6.86559345 & -1.8724586 \\ 4.12225091 & -0.92237257 & 9.49381671 & -2.37104281 \end{bmatrix}$$
$$M_2 = \begin{bmatrix} 3.24192625 & -0.39589781 & 0.40929052 & -0.45554814 \\ 6.59991996 & 4.40135399 & -3.68378817 & -2.75428419 \\ 3.31265289 & -0.99323334 & 1.59642826 & -1.05449113 \\ 6.46753364 & 2.21218758 & -2.39903301 & -1.33970849 \end{bmatrix}$$
$$M_3 = \begin{bmatrix} -21.9464225 & -21.49193731 & 33.04815917 & 4.13526652 \\ 31.71210475 & 32.02995084 & -45.84335947 & -5.30095042 \\ 36.26462491 & 34.35183299 & -50.24609045 & -5.14242071 \\ -25.9162961 & -24.95302837 & 37.98407031 & 6.16256212 \end{bmatrix}$$
$$M_4 = \begin{bmatrix} 2.08819284 & -0.60303119 & 0.32787878 & -0.08233684 \\ -0.76907073 & 3.99506513 & -0.08618573 & -1.77753553 \\ 0.29565465 & -0.86106738 & 2.09849705 & 0.67136912 \\ -0.42583356 & 0.58131019 & 0.33550322 & 0.86824497 \end{bmatrix}$$

The matrices are given as numpy-arrays below.

In [72]:

```
M1=np.array([[4.00554954, 0.01354906, 1.19192526, -1.27636247],
 [2.77659659, 1.49989982, 3.16264938, -2.48173559],
 [1.45212322, -0.48865181, 6.86559345, -1.8724586],
 [4.12225091, -0.92237257, 9.49381671, -2.37104281]])

M2=np.array([[3.24192625, -0.39589781, 0.40929052, -0.45554814],
 [6.59991996, 4.40135399, -3.68378817, -2.75428419],
 [3.31265289, -0.99323334, 1.59642826, -1.05449113],
 [6.46753364, 2.21218758, -2.39903301, -1.33970849]])

M3=np.array([[-21.9464225, -21.49193731, 33.04815917, 4.13526652],
 [31.71210475, 32.02995084, -45.84335947, -5.30095042],
 [36.26462491, 34.35183299, -50.24609045, -5.14242071],
 [-25.9162961, -24.95302837, 37.98407031, 6.16256212]])

M4=np.array([[2.08819284, -0.60303119, 0.32787878, -0.08233684],
 [-0.76907073, 3.99506513, -0.08618573, -1.77753553],
 [0.29565465, -0.86106738, 2.09849705, 0.67136912],
 [-0.42583356, 0.58131019, 0.33550322, 0.86824497]])
```

In [73]:

```
def vector_iteration(A,eps=1e-10,max_iter=100):
    """Apply vector iteration to matrix A

    The algorithm iterates until a relative tolerance `eps` is reached,
    and returns the ??? eigenvalue, corresponding eigenvector and the
    number of iterations.

    Parameters
    ----------
    A : array(N,N)
        input matrix
    eps : float
        realtive tolerance
    max_iter : int
        maximum number of iterations

    Returns
    -------
    w : float/complex
        the ??? eigenvalue of A
    v : array(N)
        corresponding eigenvector for eigenvalue w
    k : int
        number of iterations
    """

    # initialize a starting vector
    x = np.ones(len(A))

    # initialize a value for the realtive norm, the previous eigenvalue as well as the
 iteration counter
    err = 10
    counter = 0
    lambda_prev = 1

    while err > eps and counter < max_iter :

        # compute numerator
        num = np.dot(A,x)

        # compute denominator
        den = sum(abs(num)**2)**(1./2)

        # check sign
        # Not needed, cause good: see https://de.wikipedia.org/wiki/Rayleigh-Quotient

        # compute recursion
        x = num/den

        # update the relative norm
        lambda_new = den;
        err = abs(lambda_new - lambda_prev)/abs(lambda_prev)

        # increment the iteration counter
        counter += 1
        lambda_prev = lambda_new;

        # check if we exceed 75 iterations and write a message if it is the case
        if counter > 75 :
            print('Exceeded 75 iterations.')

        # DEBUGGING
        #print(num)
        #print(num)
        #print(x)
        #print()
    # end loop

    eigenvalue = np.dot(x,np.dot(A,x))/np.dot(x,x)
    eigenvector = x
    iterations = counter

    return [eigenvalue, eigenvector, iterations, err]
```

In [74]:

```
print(vector_iteration(M1))
# w,v = np.linalg.eig(M1)
# print(w,'\n')

print(vector_iteration(M2))
# w,v = np.linalg.eig(M2)
# print(w,'\n')

print(vector_iteration(M3))
# w,v = np.linalg.eig(M3)
# print(w,'\n')

print(vector_iteration(M4))
# w,v = np.linalg.eig(M4)
# print(w,'\n')
```

```
[4.000000006767723, array([-0.55622662, -0.43458467,  0.52119876,  0.47968
737]), 70, 9.635381363729208e-11]
Exceeded 75 iterations.
[4.0000000043482675, array([-0.51263192,  0.32656906, -0.78269758, -0.1339
6141]), 76, 7.533085257504564e-11]
[-39.99999999955286, array([ 0.39076875, -0.5298475 , -0.612742  ,  0.4371
5975]), 12, 2.1623769442377138e-11]
[3.9999999991408677, array([ 0.33873297, -0.84597746,  0.37836296, -0.1625
5332]), 34, 7.170797290047376e-11]
```

FE-Matrices

Load the Matrices

Load the system matrices. The matices are real, square and symmetric with dimension $3N \times 3N$. The DoFs are arranged in the order $x_1, y_1, z_1, x_2, \ldots, z_N$ where x_i denotes the x-displacement of node i.

In [75]:

```
M = mmread('Ms.mtx').toarray() # mass matrix
K = mmread('Ks.mtx').toarray() # stiffness 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

# Debugging
# print(M)
```

The DoFs in the system matrices are arranged according to a regular grid of linear finite elements. In the following we determine the unique x, y, and z coodinates of the grid.

In [76]:

```
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))
print('Nx =',len(x))
print('Ny =',len(y))
print('Nz =',len(z))
# 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)])
# or equivalent: Xg,Yg,Zg  = np.meshgrid(x,y,z)
```

```
Nx = 28
Ny = 16
Nz = 5
```

Plot the Geometry

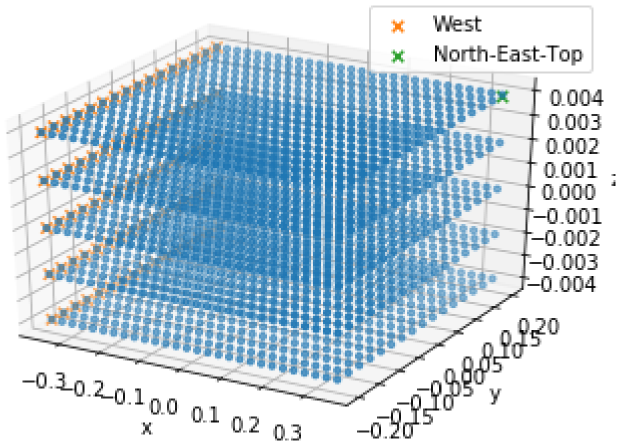
One can plot the location of the nodes, select subsets of nodes and plot them ...

In [77]:

```
# plot the geometric points
from mpl_toolkits.mplot3d import Axes3D
fig,ax = plt.subplots(subplot_kw={'projection':'3d'})

#sm = 0.1/mode.max()
ax.scatter(X[:,0],X[:,1],X[:,2],s=10)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

# select nodes on the west-side, i.e. at x=x_min
tol = 1e-12
x_min = X[:,0].min()
Nw = np.argwhere(np.abs(X[:,0]-x_min)<tol) # Node indices of West-Edge nodes
# select node on North-East-Top corner
Nnet = np.argwhere(np.all(np.abs(X-X.max(axis=0))<tol,axis=1))[0]

ax.scatter(X[Nw,0],X[Nw,1],X[Nw,2],s=30,marker='x',label='West')
ax.scatter(X[Nnet,0],X[Nnet,1],X[Nnet,2],s=30,marker='x',label='North-East-Top')
ax.legend()
```

Out[77]:

```
<matplotlib.legend.Legend at 0x22d1458c688>
```



Solve a Static Problem

Solve a static problem applying nodal forces to the North-East-Top corner and fixing all DoF at the West-Edge of the plate.

We solve the system

$$Ku = f$$

for the displacements u. The system needs to be constrained, thus, we select nodes which will be removed from the system.

In [78]:

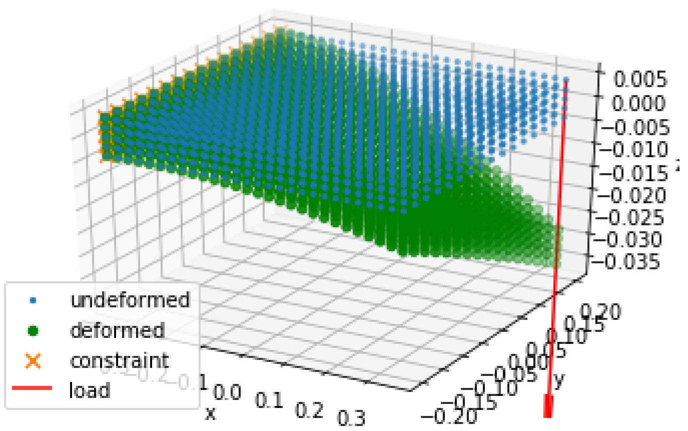
```
# because the dofs are ordered as x_1, y_1, z_1, x_2, ..., z_N in the global system, th
e x, y, and z dofs for node n are
# located at position 3n, 3n+1, 3n+2.

# 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[Nw],Iy[Nw],Iz[Nw]]).ravel() # dof indices of fix constraint
Ic = np.array([(i in If) for i in np.arange(3*N)]) # boolean array of constraind dofs

# construct forcing vector
f = np.zeros(3*N)
f[Iz[Nnet]] = -1.0

# compute the reduced system
Kc = K[np.ix_(~Ic,~Ic)]
fc = f[~Ic]

# compute solution
u = np.zeros(3*N) # initialize displacement vector

# solve the linear system Kc*uc=fc
uc = np.linalg.solve(Kc,fc)

# sort solution in large vector
u[~Ic] = uc
```

In [79]:

```
# plot in 3D
fig,ax = plt.subplots(subplot_kw={'projection':'3d'})

#sm = 0.1/mode.max()
ax.scatter(X[:,0],X[:,1],X[:,2],s=5,label='undeformed') # undeformed

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

# scale factor for plotting
s = 0.5/np.max(np.sqrt(np.sum(U**2,axis=0)))
Xu = X + s*U # defomed configuration (displacement scaled by s)

ax.scatter(Xu[:,0],Xu[:,1],Xu[:,2],c='g',label='deformed')
ax.scatter(X[Nw,0],X[Nw,1],X[Nw,2],s=50,marker='x',label='constraint')
ax.quiver(X[:,0],X[:,1],X[:,2],f[Ix],f[Iy],f[Iz],color='r',length=0.1,label='load')

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

Out[79]:

```
<matplotlib.legend.Legend at 0x22d15e0f248>
```

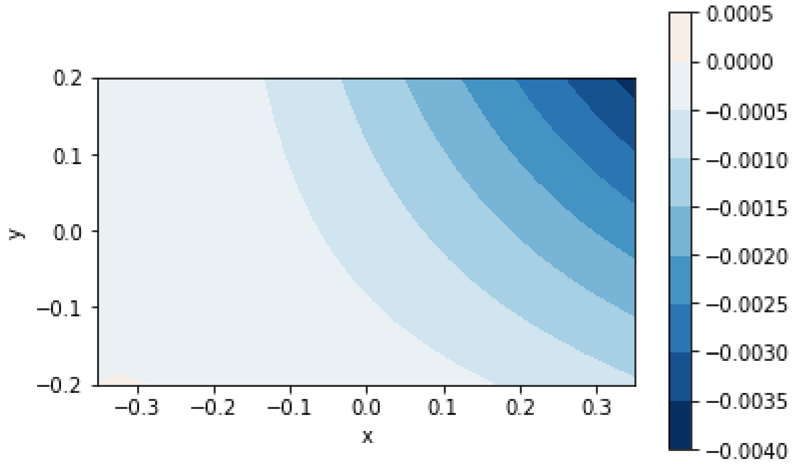


In [80]:

```
# plot in 2D (z-displacement of the top-nodes)

# select nodes
Nt = np.argwhere(np.abs(X[:,2]-X[:,2].max())<tol)
# extract z-displacements
uz = np.reshape(u[Iz[Nt]],[len(y),len(x)])

lim = np.max(np.abs(uz)) # limit to center color legend around 0

fig,ax = plt.subplots()
cax = ax.contourf(x,y,uz,cmap=plt.get_cmap('RdBu_r'),vmin=-lim,vmax=lim)
fig.colorbar(cax,extend='both')#,orientation='horizontal')
ax.set_aspect('equal')
ax.set_xlabel('x')
ax.set_ylabel('y')
```

Out[80]:

```
Text(0, 0.5, 'y')
```



Submission Task 2: Compute eigenvalues and modeshapes

All free

In the first step, use the unconstrained system to compute the first 15 eigenmodes and plot them. In order to compute only the first k modes use the code

```
# only compute a subset of modes
from scipy.linalg import eigh
k = 15
W,V = eigh(K,M,eigvals=(0,k))
```

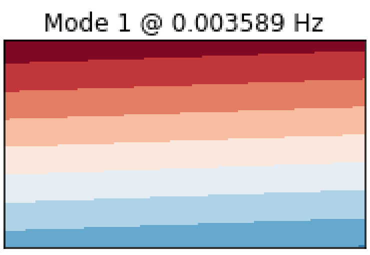
for the plots of the eigenmodes you can use the function given below.

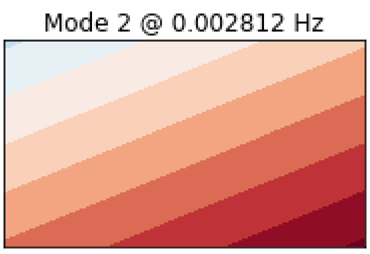
In [90]:

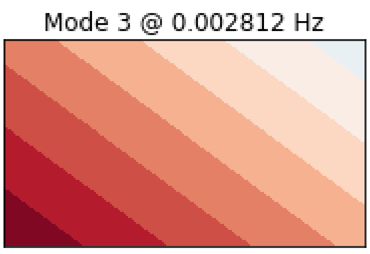
```python
def plotmodes(V_var,W_var) :
    for i,v in enumerate(V_var.T) : # iterate over eigenvectors
        c = np.reshape(v[Iz[Nt]],[len(y),len(x)])
        lim = np.max(np.abs(c))
        fig,ax = plt.subplots(figsize=[3.5,2])
        ax.contourf(x,y,c,cmap=plt.get_cmap('RdBu'),vmin=-lim,vmax=lim)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


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

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

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

        # scale factor for plotting
        s = 0.5/np.max(np.sqrt(np.sum(U**2,axis=0)))
        Xu = X + s*U # defomed configuration (displacement scaled by s)

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

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

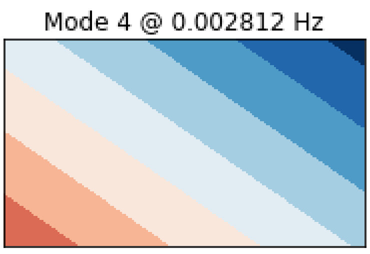
In [91]:

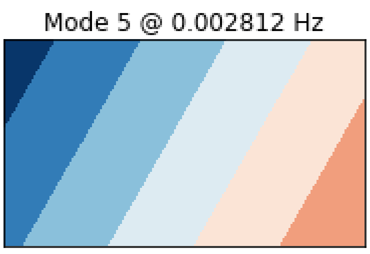
```
# only compute a subset of modes
from scipy.linalg import eigh
k = 15
W,V = eigh(K,M,eigvals=(0,k))

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

# do it fancier
Ic_false = np.zeros((3*N),dtype=bool) #vector containing falses
N_con_flase = [];
makeFancyModes(V,Ic_false,W,N_con_flase)
```

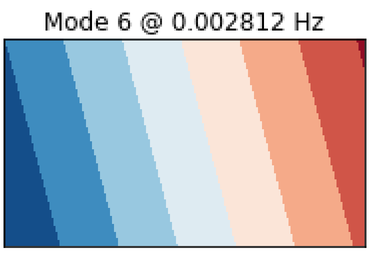

Mode 1 @ 0.003589 Hz


Mode 2 @ 0.002812 Hz

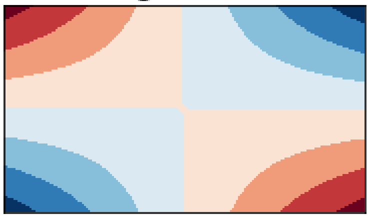

Mode 3 @ 0.002812 Hz


Mode 4 @ 0.002812 Hz

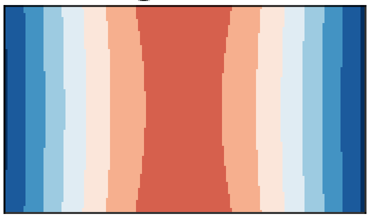

Mode 5 @ 0.002812 Hz


Mode 6 @ 0.002812 Hz

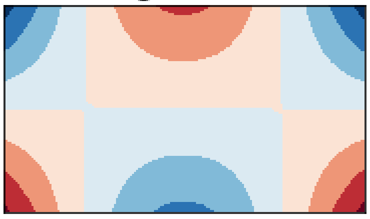
Mode 7 @ 12.828632 Hz



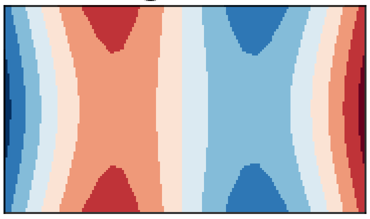
Mode 8 @ 26.527991 Hz



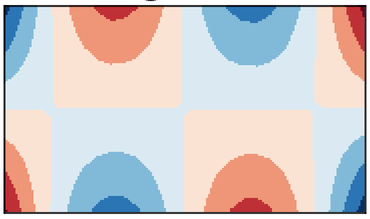
Mode 9 @ 37.295051 Hz



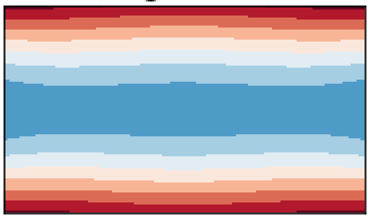
Mode 10 @ 73.160914 Hz



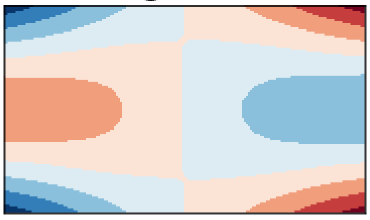
Mode 11 @ 83.196980 Hz

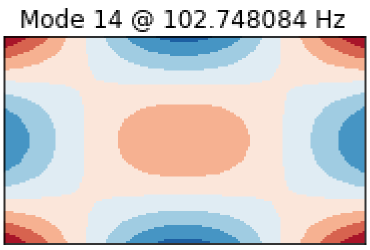


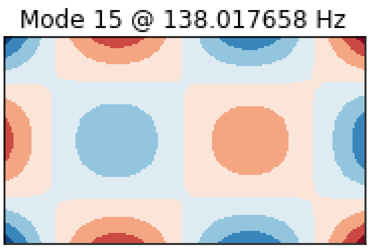
Mode 12 @ 83.485626 Hz

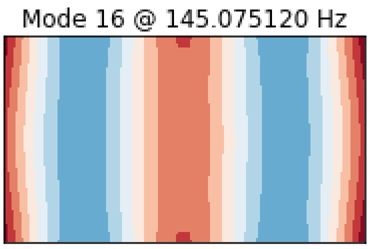


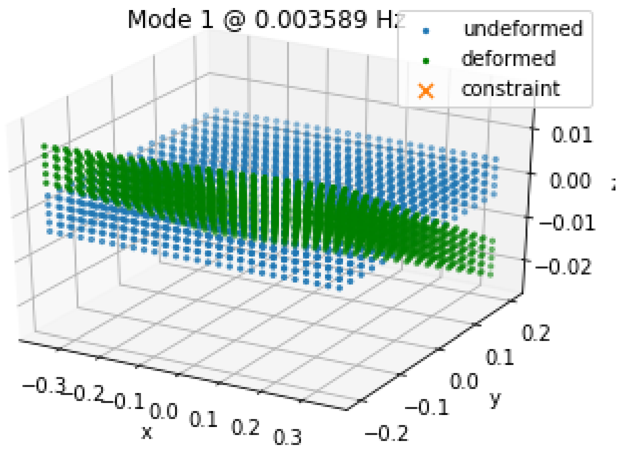
Mode 13 @ 87.724098 Hz

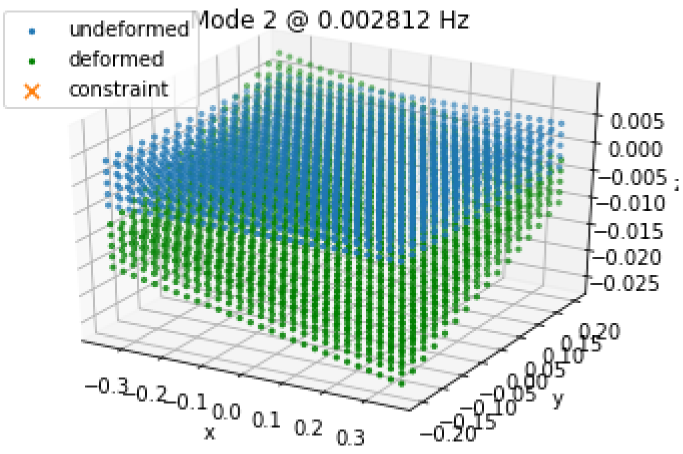


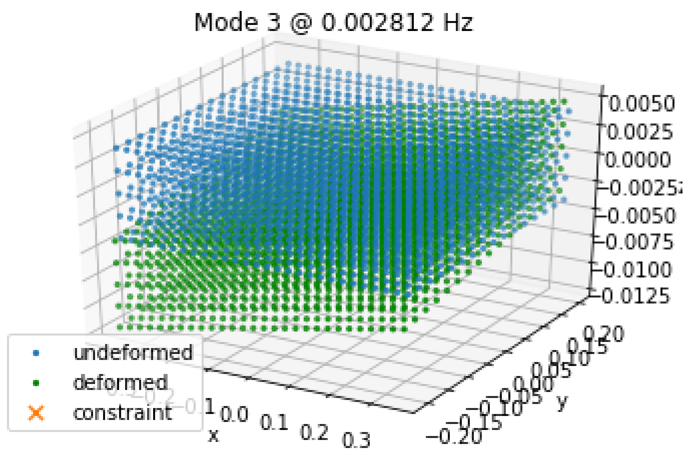

Mode 14 @ 102.748084 Hz

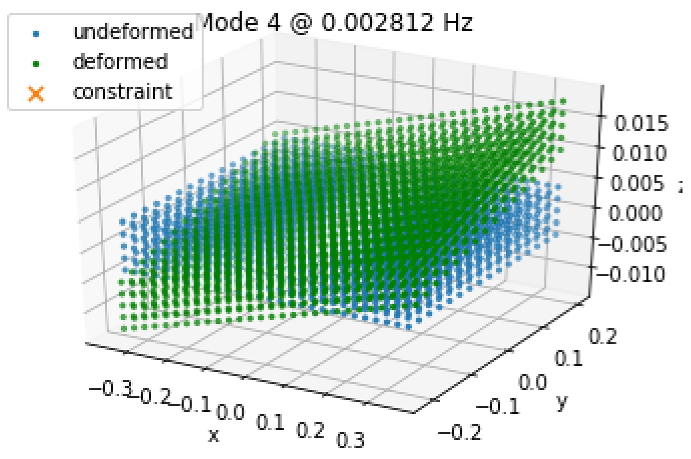

Mode 15 @ 138.017658 Hz

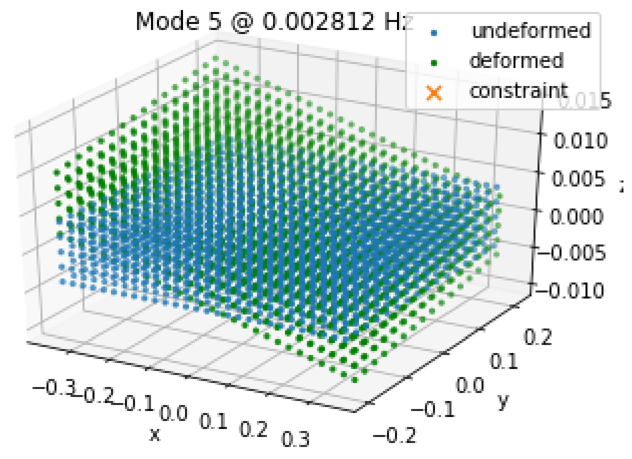

Mode 16 @ 145.075120 Hz

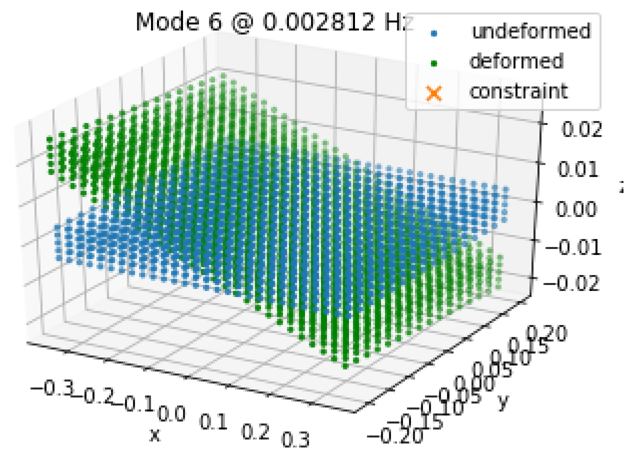

Mode 1 @ 0.003589 Hz
undeformed
deformed
constraint
0.01
0.00
−0.01
−0.02
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
−0.2
−0.1
0.0
0.1
0.2
y

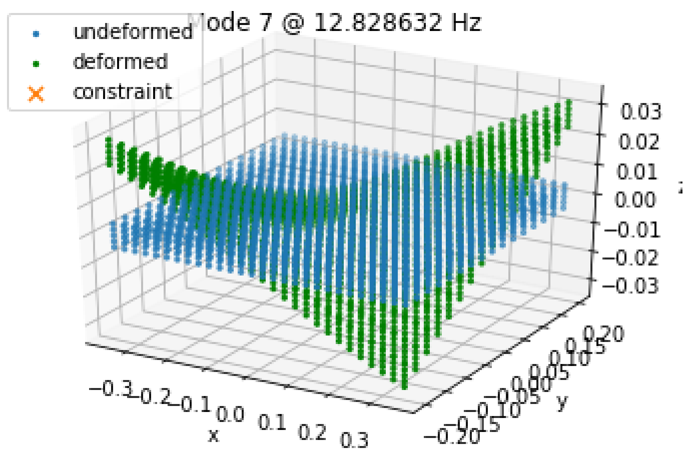

Mode 2 @ 0.002812 Hz
undeformed
deformed
constraint
0.005
0.000
−0.005
−0.010
−0.015
−0.020
−0.025
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
y

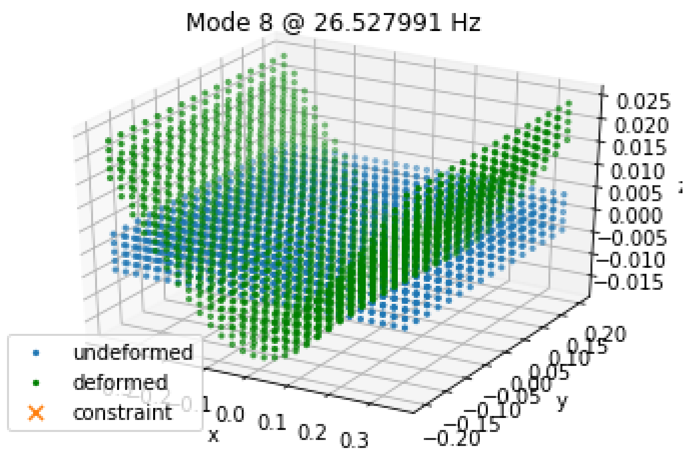

Mode 3 @ 0.002812 Hz
undeformed
deformed
constraint
x
y
0.0050
0.0025
0.0000
−0.0025
−0.0050
−0.0075
−0.0100
−0.0125

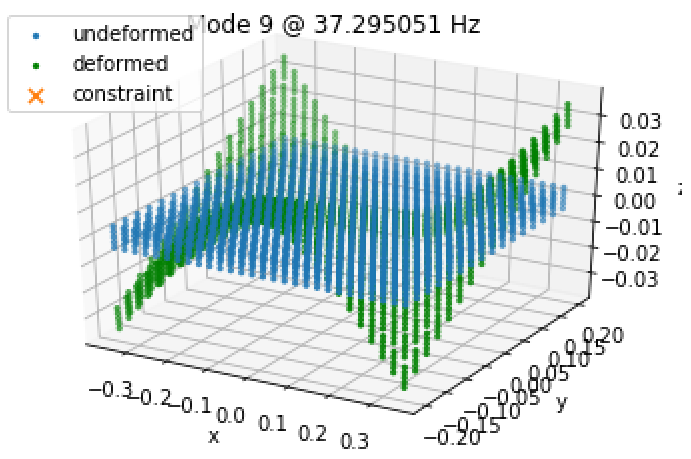

Mode 4 @ 0.002812 Hz
undeformed
deformed
constraint
x
y
0.015
0.010
0.005
0.000
−0.005
−0.010

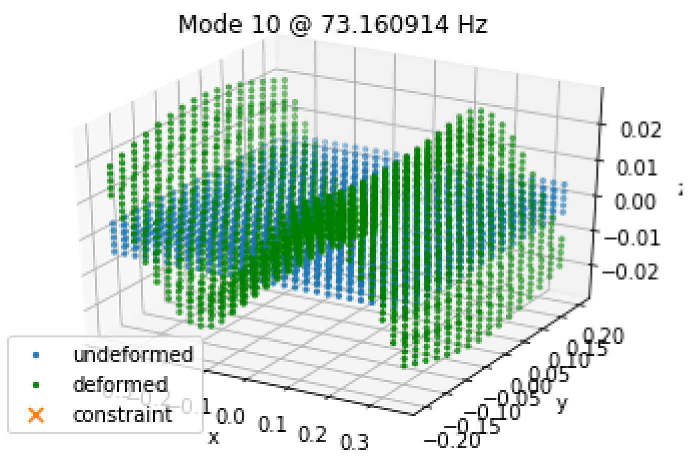

Mode 5 @ 0.002812 Hz
undeformed
deformed
constraint
x
y
0.010
0.005
0.000
−0.005
−0.010

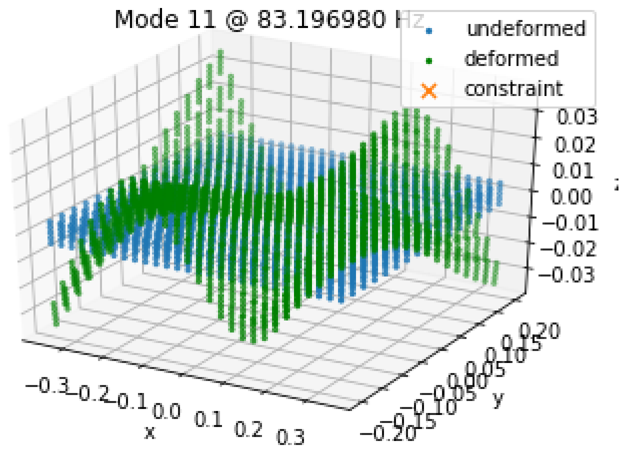

Mode 6 @ 0.002812 Hz
undeformed
deformed
constraint
x
y
0.02
0.01
0.00
−0.01
−0.02

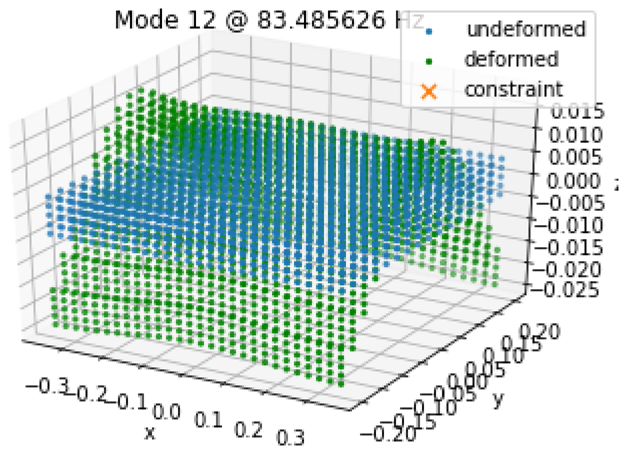

Mode 7 @ 12.828632 Hz
undeformed
deformed
constraint
x
y
z

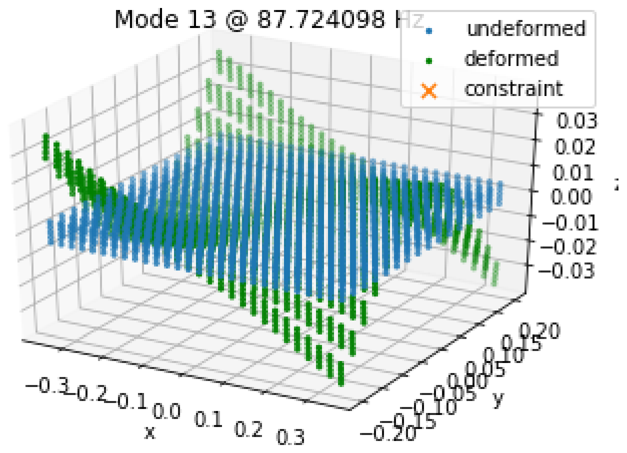

Mode 8 @ 26.527991 Hz
undeformed
deformed
constraint
x
y
z

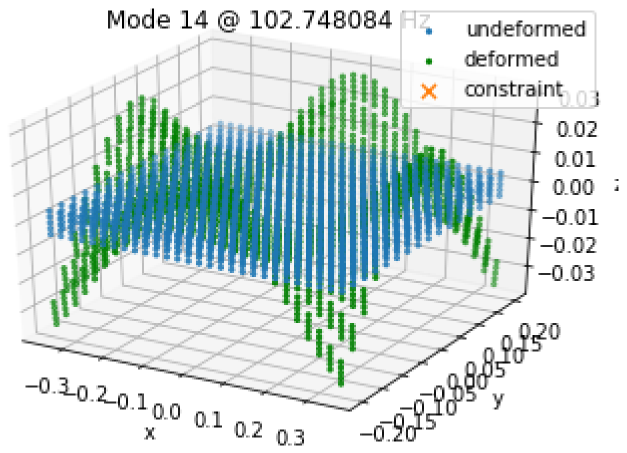

Mode 9 @ 37.295051 Hz
undeformed
deformed
constraint
x
y
z

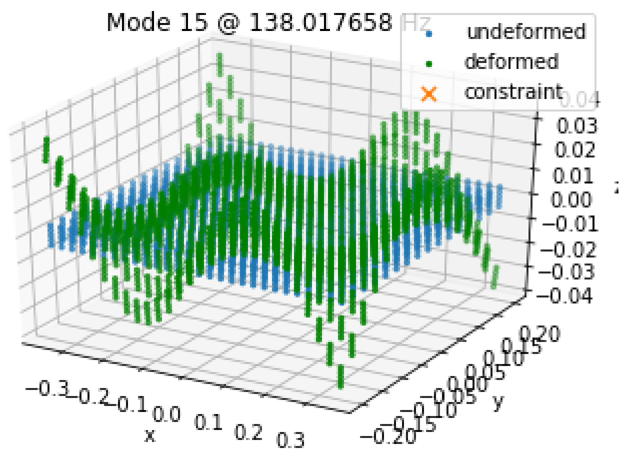

Mode 10 @ 73.160914 Hz
undeformed
deformed
constraint
x
y
z

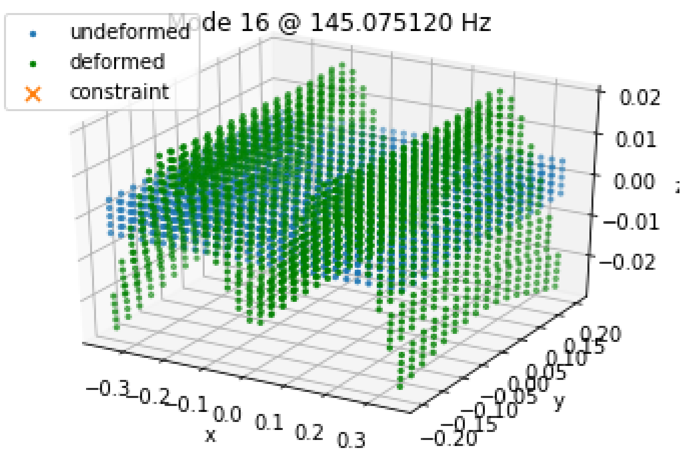

Mode 11 @ 83.196980 Hz
undeformed
deformed
constraint
x
y
z


Mode 12 @ 83.485626 Hz
undeformed
deformed
constraint
x
y
z


Mode 13 @ 87.724098 Hz
undeformed
deformed
constraint
x
y
z


Mode 14 @ 102.748084 Hz
undeformed
deformed
constraint
x
y
z



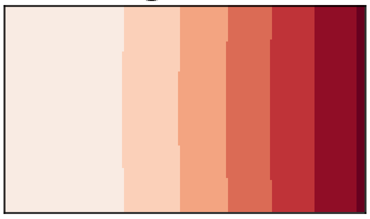


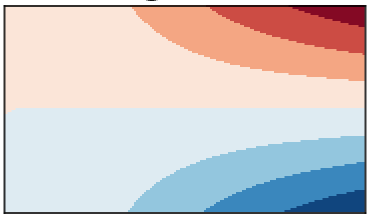
Short side clamped

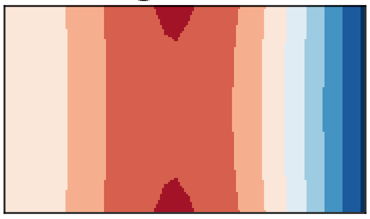
In the next step constrain the nodes of the short edge on the north-west side (Nw), compute the first 15 eigenmodes and plot them like in the previous task.

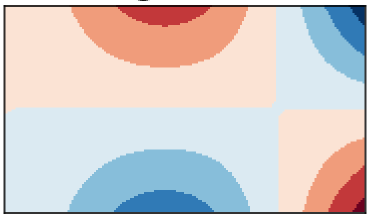
In [92]:

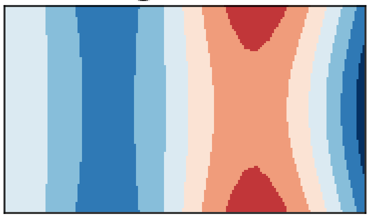
```
# compute the reduced model based on previous calculations of Nw
Mc = M[np.ix_(~Ic,~Ic)]
Kc = K[np.ix_(~Ic,~Ic)]

# only compute a subset of modes of the reduced model
k = 15
W,V = eigh(Kc,Mc,eigvals=(0,k))

# add missing nodes (constraints)
V_new = np.zeros((len(Iz)*3,k+1))
If_sort = np.sort(If);
for i,v in enumerate(V.T):
    V_dat = V[:,i]
    for d,idx in enumerate(If_sort) :
        V_dat = np.insert(V_dat,idx,0)
    V_new[:,i] = V_dat

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

# # do it fancier
makeFancyModes(V_new,Ic,W,Nw)
```

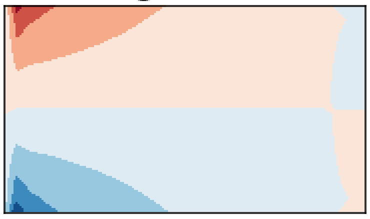

Mode 1 @ 4.187423 Hz


Mode 2 @ 8.572591 Hz

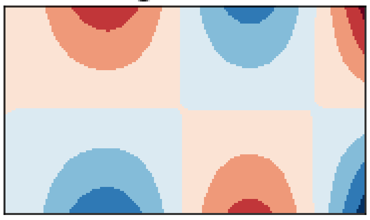

Mode 3 @ 26.246231 Hz


Mode 4 @ 33.737415 Hz

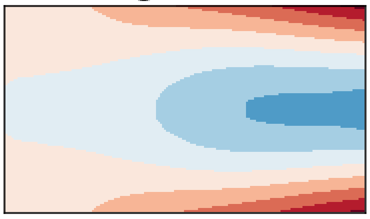

Mode 5 @ 73.567618 Hz


Mode 6 @ 77.981787 Hz

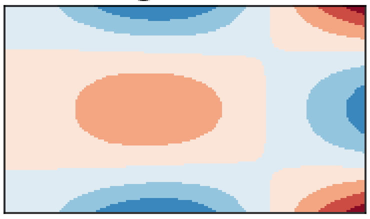
Mode 7 @ 80.840503 Hz



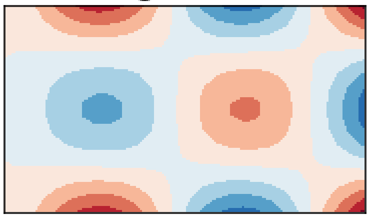
Mode 8 @ 84.977848 Hz



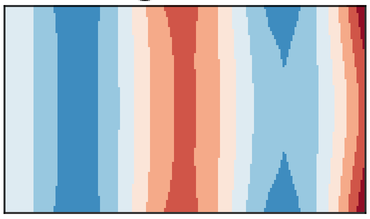
Mode 9 @ 97.703477 Hz



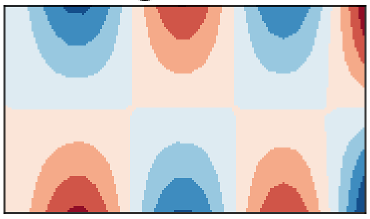
Mode 10 @ 131.745310 Hz

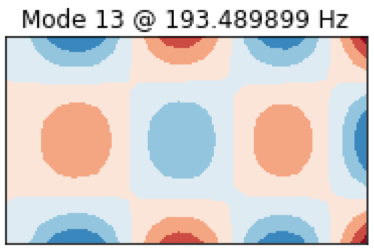


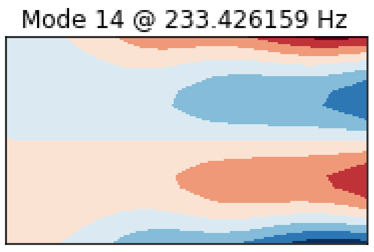
Mode 11 @ 145.399305 Hz

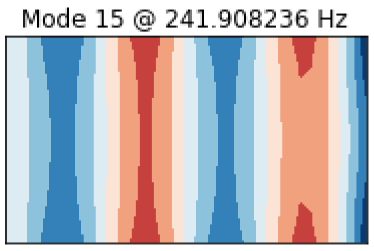


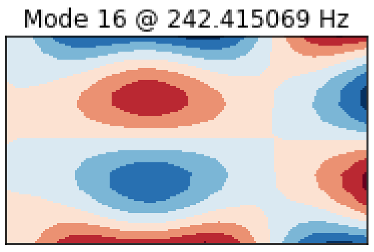
Mode 12 @ 152.040222 Hz

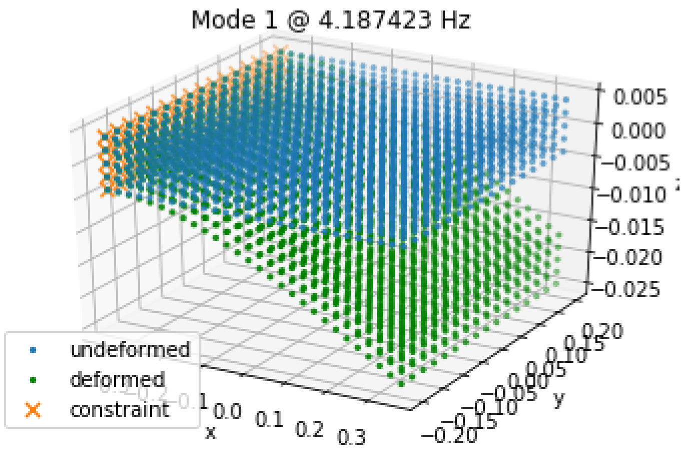


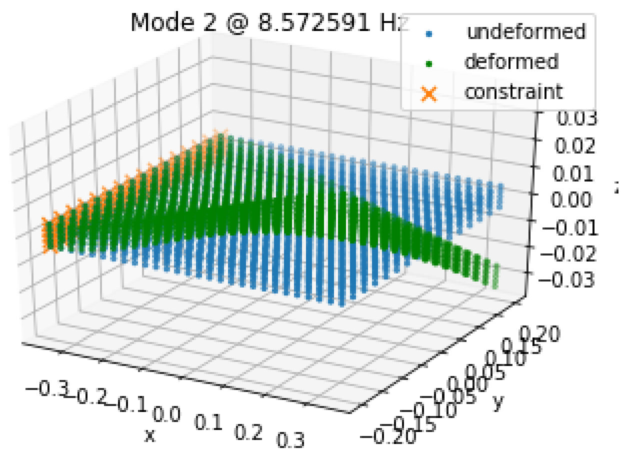

Mode 13 @ 193.489899 Hz

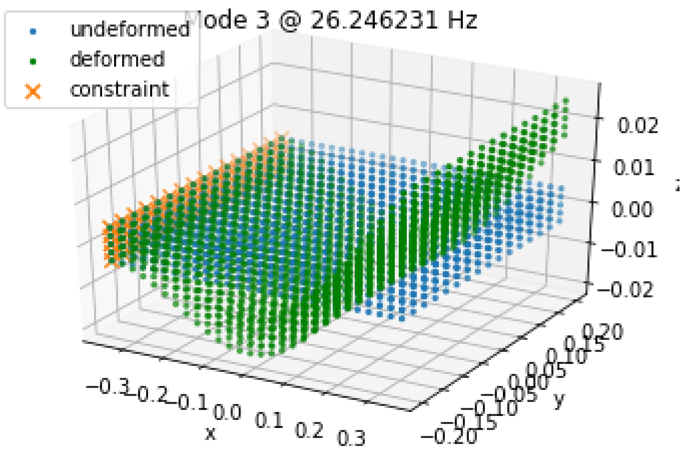

Mode 14 @ 233.426159 Hz

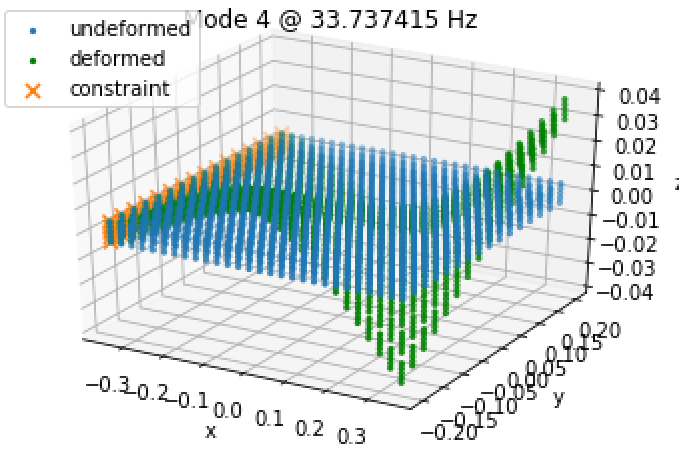

Mode 15 @ 241.908236 Hz

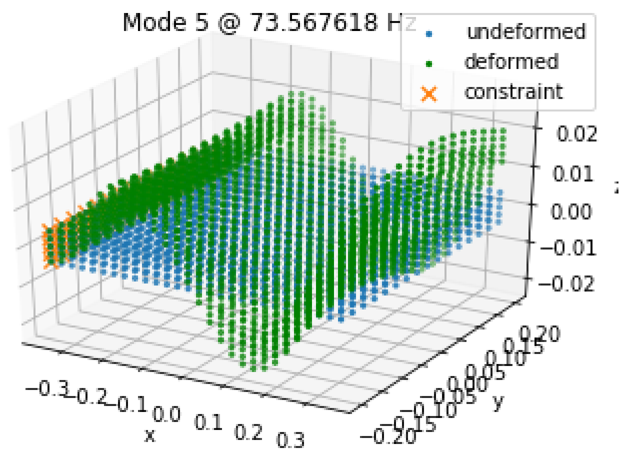

Mode 16 @ 242.415069 Hz

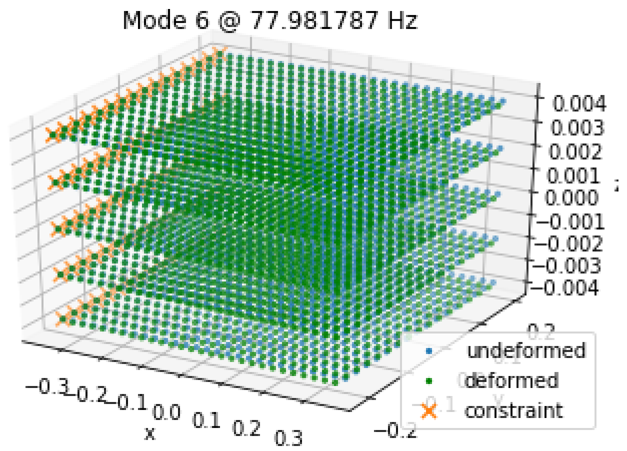

Mode 1 @ 4.187423 Hz
0.005
0.000
−0.005
−0.010
−0.015
−0.020
−0.025
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
y
0.0
0.1
0.2
0.3
x
undeformed
deformed
constraint

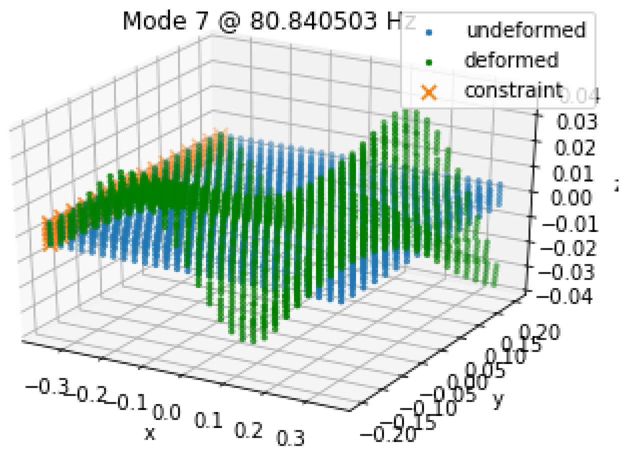

Mode 2 @ 8.572591 Hz
undeformed
deformed
constraint
x
y
z

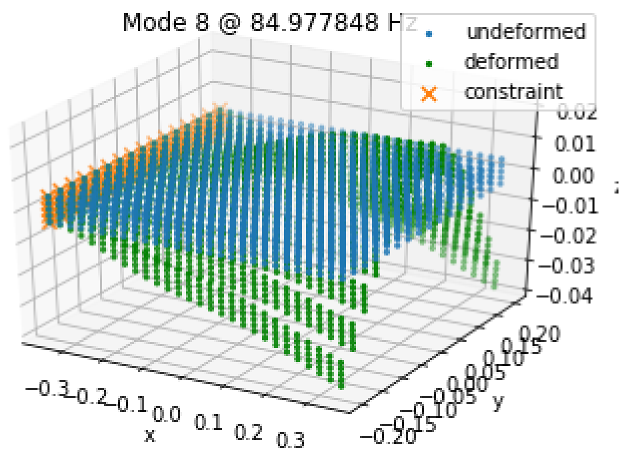

Mode 3 @ 26.246231 Hz
undeformed
deformed
constraint
x
y
z

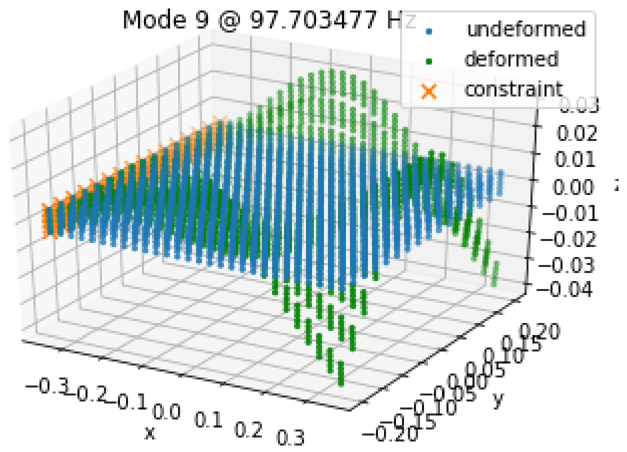

Mode 4 @ 33.737415 Hz
undeformed
deformed
constraint
x
y
z

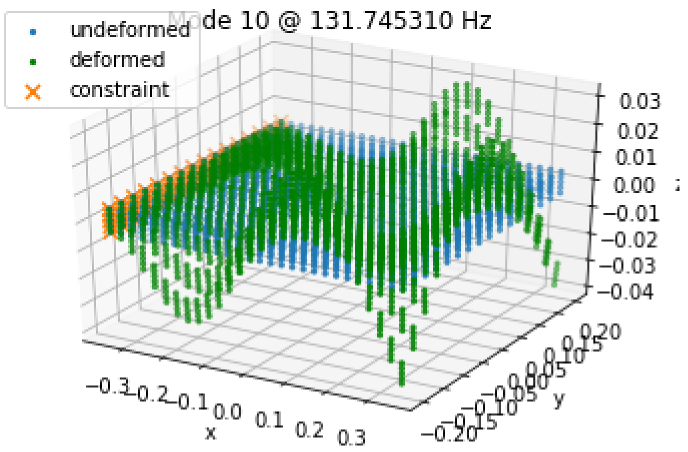

Mode 5 @ 73.567618 Hz
undeformed
deformed
constraint
x
y
z

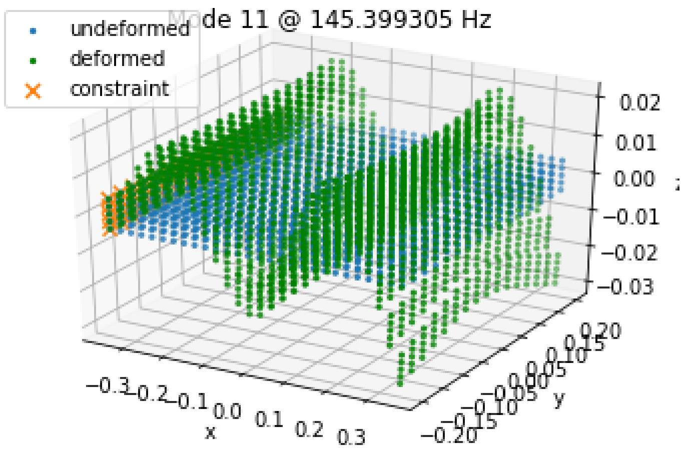

Mode 6 @ 77.981787 Hz
undeformed
deformed
constraint
x
y
z

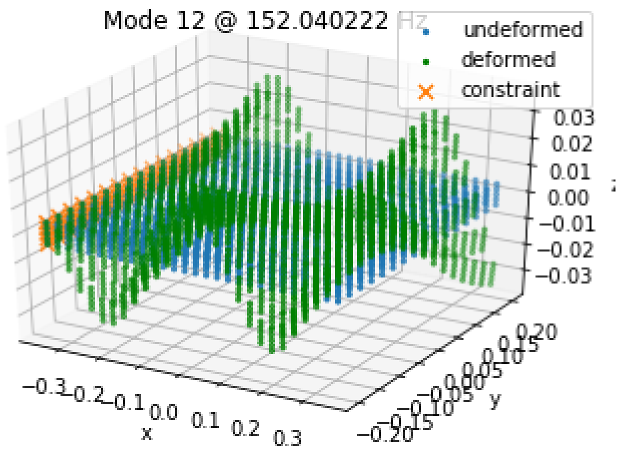

Mode 7 @ 80.840503 Hz
undeformed
deformed
constraint
x
y
z

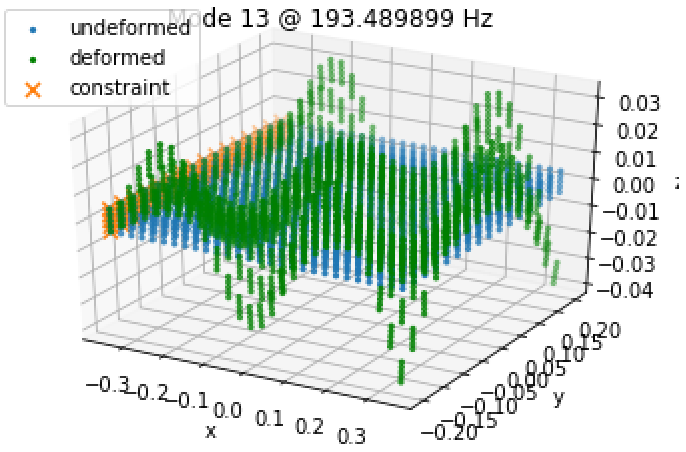

Mode 8 @ 84.977848 Hz
undeformed
deformed
constraint
x
y
z

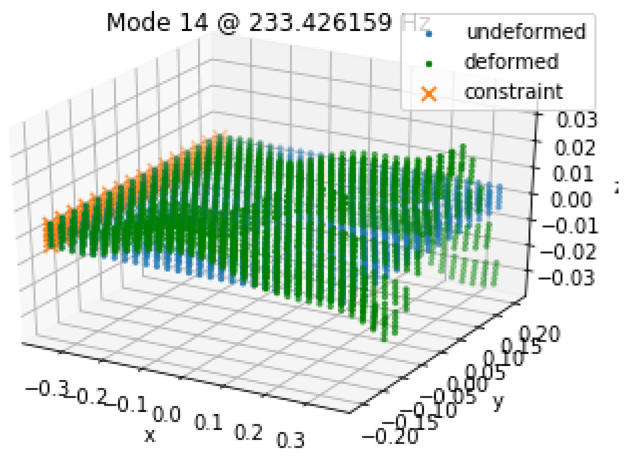

Mode 9 @ 97.703477 Hz
undeformed
deformed
constraint
x
y
z

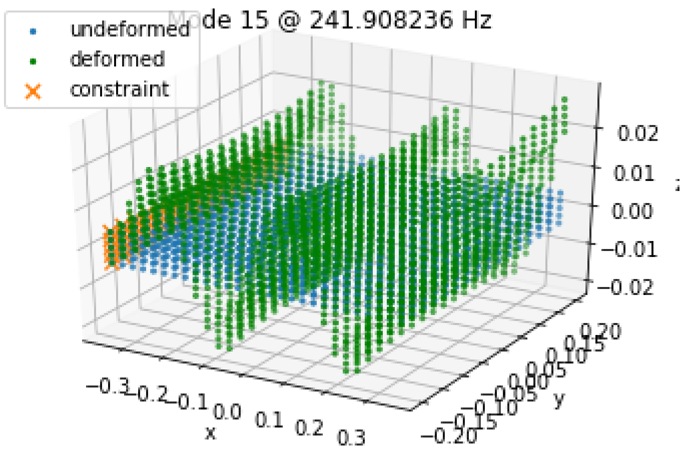

Mode 10 @ 131.745310 Hz
undeformed
deformed
constraint
x
y
z

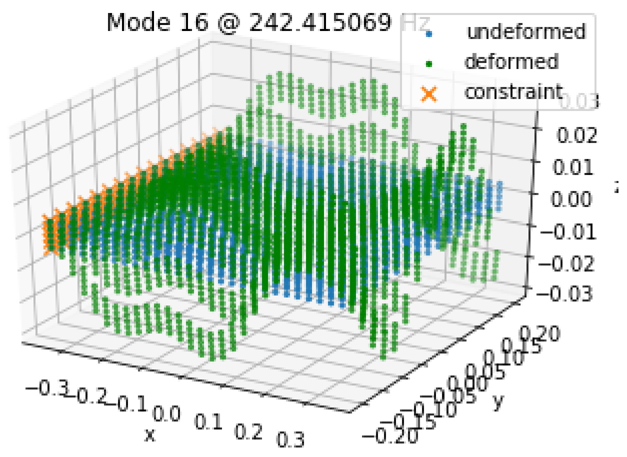

Mode 11 @ 145.399305 Hz
undeformed
deformed
constraint
x
y
z


Mode 12 @ 152.040222 Hz
undeformed
deformed
constraint
x
y
z


Mode 13 @ 193.489899 Hz
undeformed
deformed
constraint
x
y
z







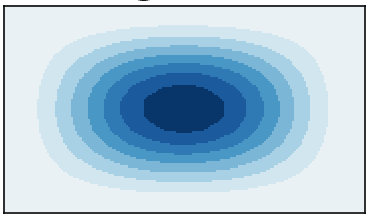
Clamped edges

In the last step all edges of the plate should be clamped (Nn, No, Ns, Nw). Compute the first 15 eigenmodes and plot them like in the previous tasks.

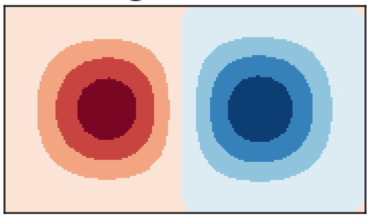
In [93]:

```
tol = 1e-12
# select nodes on the North-side, i.e. at y=y_max
y_max = X[:,1].max()
Nn = np.argwhere(np.abs(X[:,1]-y_max)<tol) # Node indices of North-Edge nodes

# select nodes on the East-side, i.e. at x=x_max
x_max = X[:,0].max()
No = np.argwhere(np.abs(X[:,0]-x_max)<tol) # Node indices of East-Edge nodes

# select nodes on the South-side, i.e. at y=y_min
y_min = X[:,1].min()
Ns = np.argwhere(np.abs(X[:,1]-y_min)<tol) # Node indices of South-Edge nodes

N_con = np.unique(np.concatenate((Nn,No,Ns,Nw))) #concatenate all and only take unique
 (remove the double ones)
# # modify constraints: select which indices in the global system must be constrained
If_all = np.array([Ix[Ncon],Iy[Ncon],Iz[Ncon]]).ravel() # dof indices of fix constraint
Ic_all = np.array([(i in If_all) for i in np.arange(3*N)]) # boolean array of constrain
d dofs

# compute the reduced model based on previous calculations of Nw
Mc_all = M[np.ix_(~Ic_all,~Ic_all)]
Kc_all = K[np.ix_(~Ic_all,~Ic_all)]

# only compute a subset of modes of the reduced model
k = 15
W,V = eigh(Kc_all,Mc_all,eigvals=(0,k))

# add missing nodes (constraints)
V_new = np.zeros((len(Iz)*3,k+1))
If_sort_all = np.sort(If_all);
for i,v in enumerate(V.T):
    V_dat = V[:,i]
    for d,idx in enumerate(If_sort_all) :
        V_dat = np.insert(V_dat,idx,0)
    V_new[:,i] = V_dat

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

# do it fancier
makeFancyModes(V_new,Ic_all,W,N_con)
```

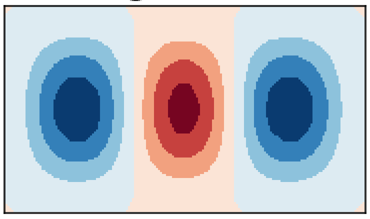
Mode 1 @ 90.276877 Hz



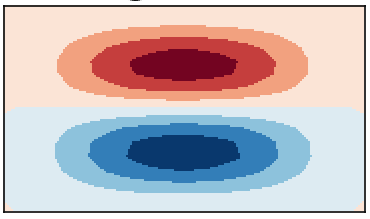
Mode 2 @ 117.317754 Hz



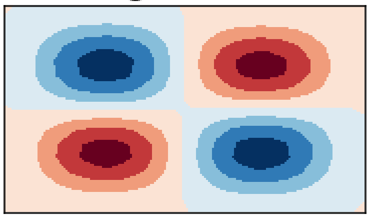
Mode 3 @ 175.695341 Hz



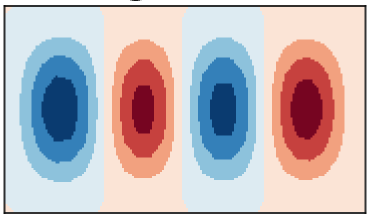
Mode 4 @ 238.501830 Hz



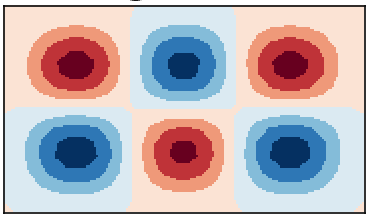
Mode 5 @ 254.410780 Hz



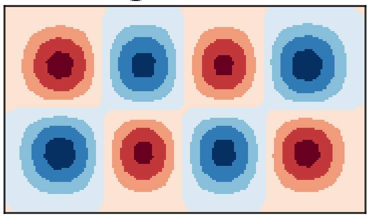
Mode 6 @ 265.083064 Hz



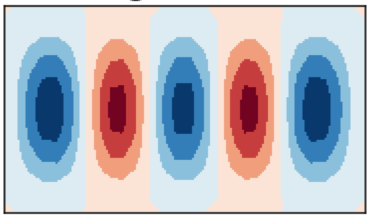
Mode 7 @ 292.082299 Hz



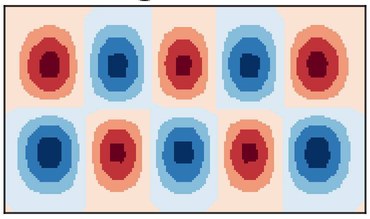
Mode 8 @ 359.702453 Hz



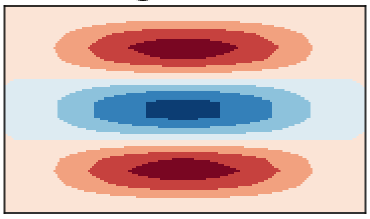
Mode 9 @ 383.695469 Hz



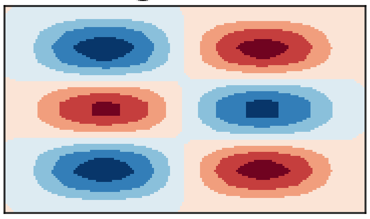
Mode 10 @ 460.570309 Hz



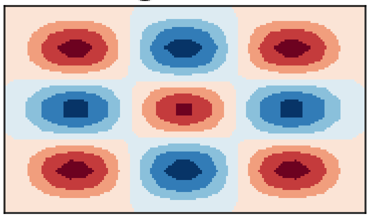
Mode 11 @ 469.752586 Hz



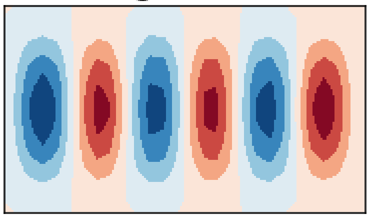
Mode 12 @ 481.446785 Hz



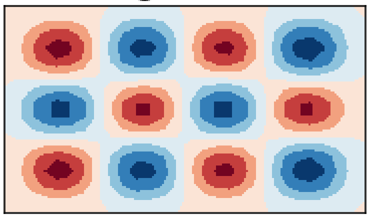
Mode 13 @ 507.539465 Hz



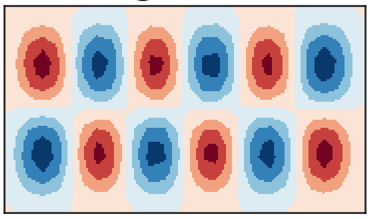
Mode 14 @ 531.219220 Hz



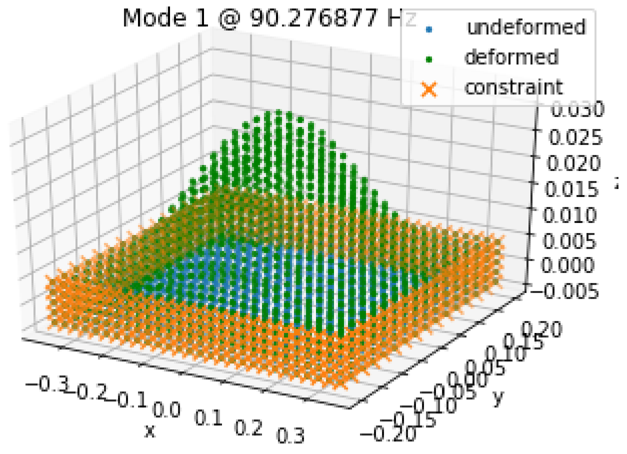
Mode 15 @ 555.617092 Hz

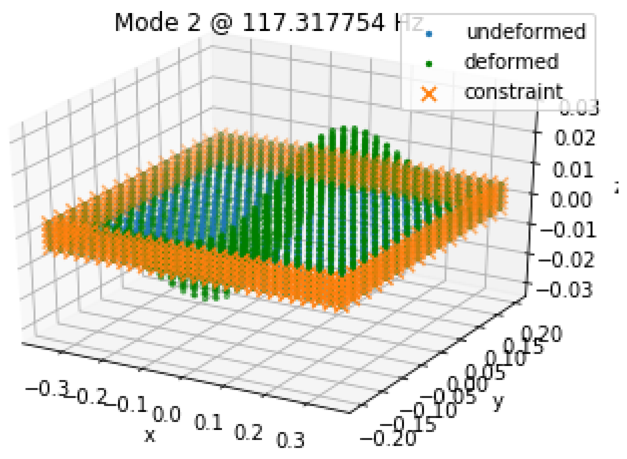


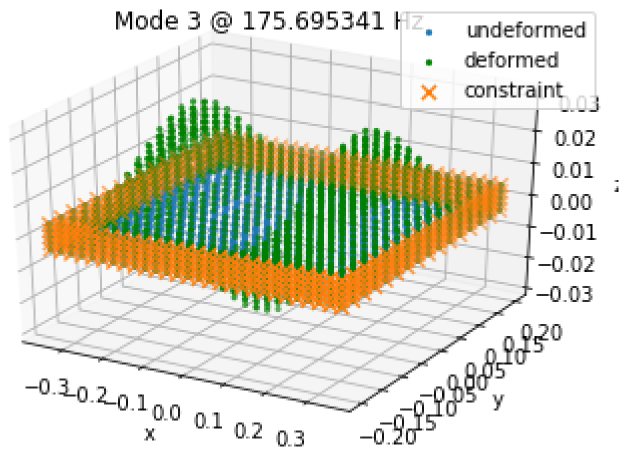
Mode 16 @ 594.892648 Hz

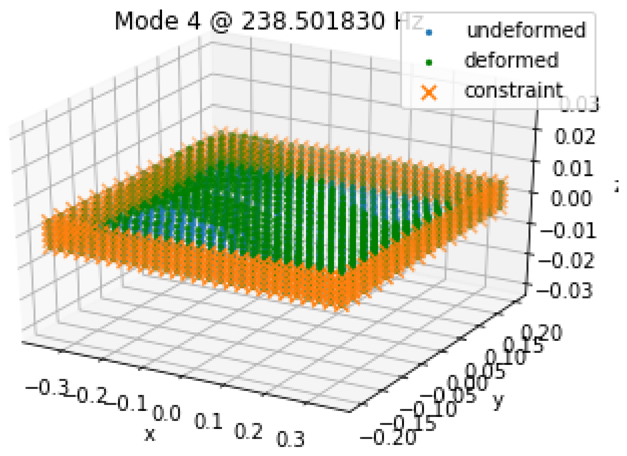


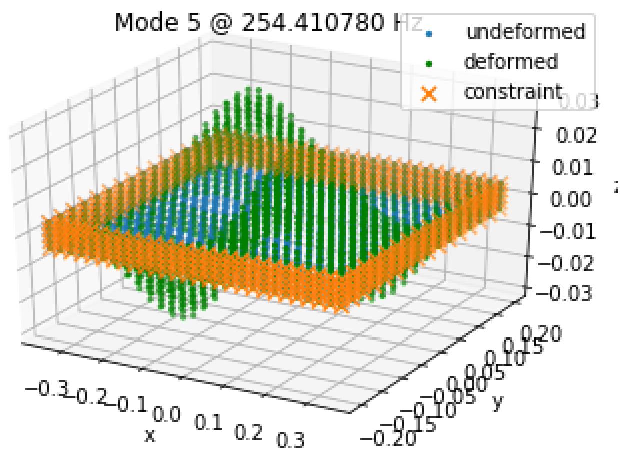
Mode 1 @ 90.276877 Hz

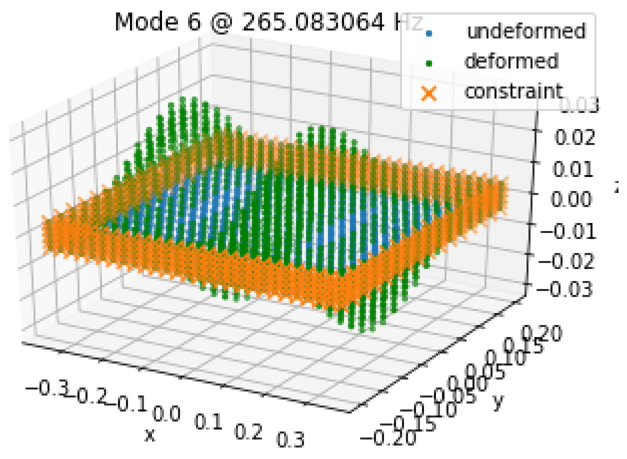


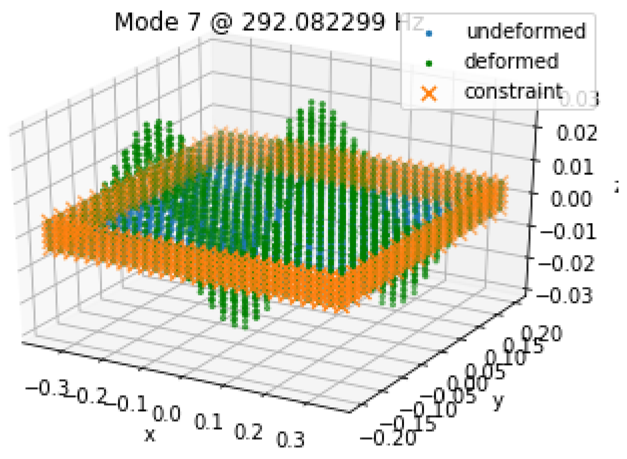

Mode 2 @ 117.317754 Hz
undeformed
deformed
constraint
x
y
z

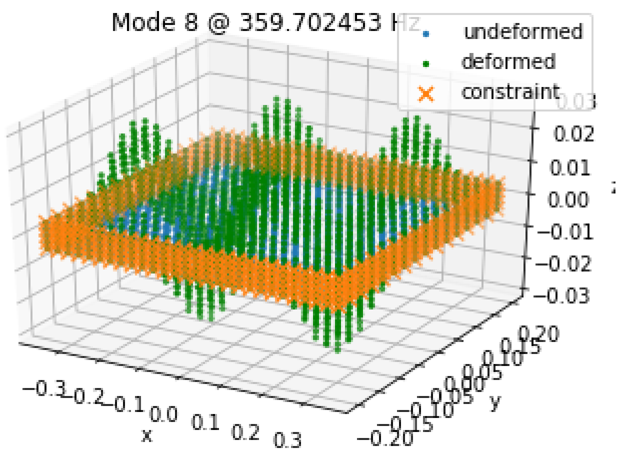

Mode 3 @ 175.695341 Hz
undeformed
deformed
constraint
x
y
z

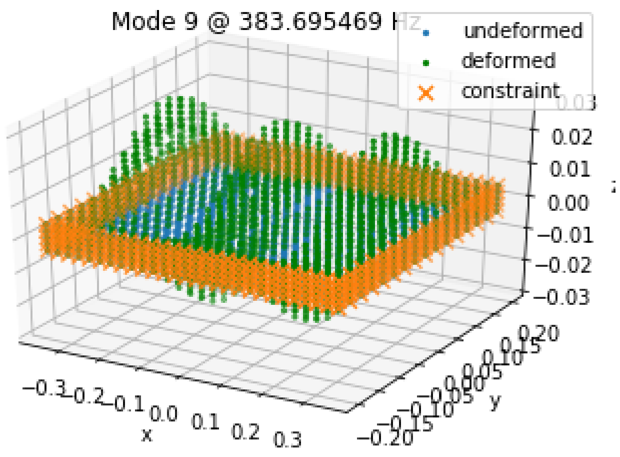

Mode 4 @ 238.501830 Hz
undeformed
deformed
constraint
x
y
z

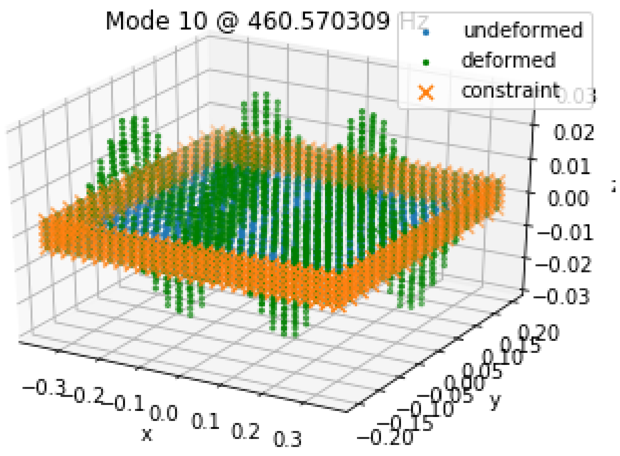

Mode 5 @ 254.410780 Hz
undeformed
deformed
constraint
x
y
z

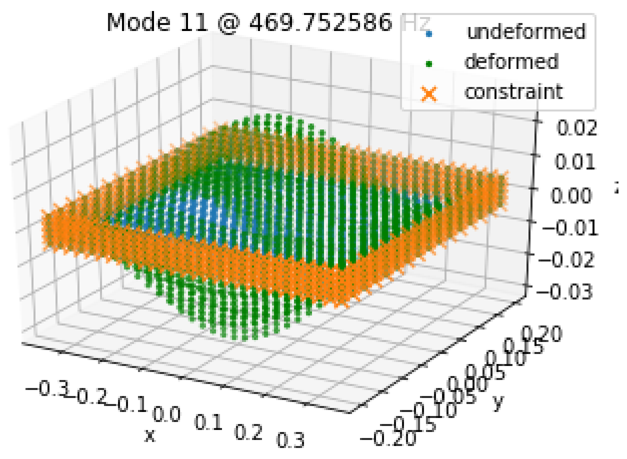

Mode 6 @ 265.083064 Hz
undeformed
deformed
constraint
x
y
z

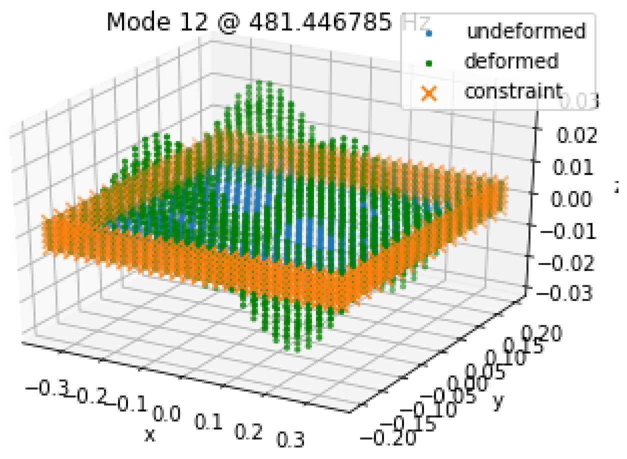

Mode 7 @ 292.082299 Hz
undeformed
deformed
constraint
x
y
z

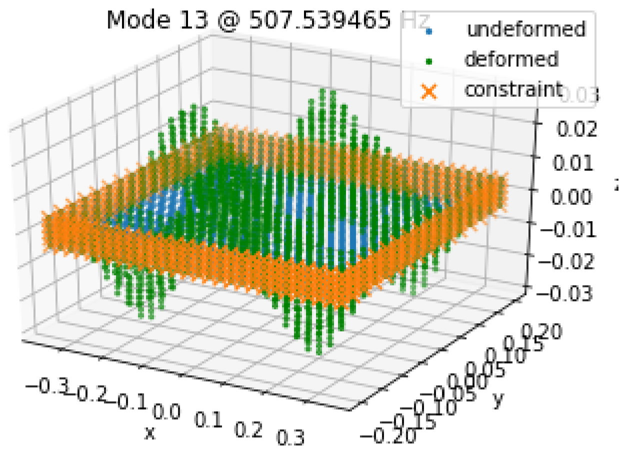

Mode 8 @ 359.702453 Hz
undeformed
deformed
constraint
x
y
z

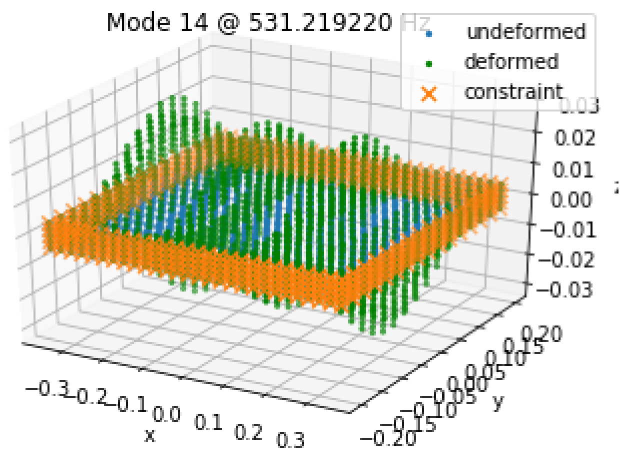

Mode 9 @ 383.695469 Hz
undeformed
deformed
constraint
x
y
z

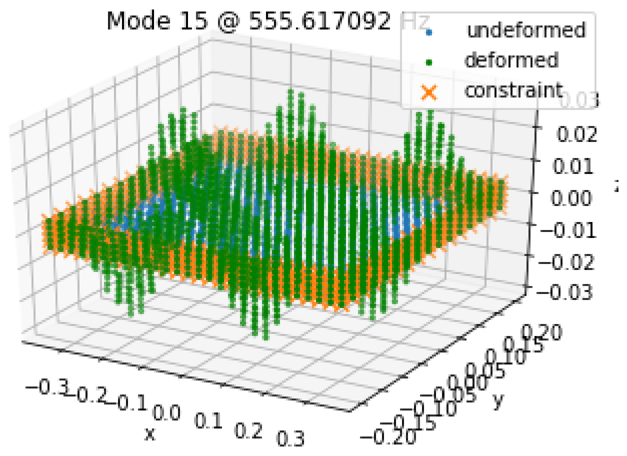

Mode 10 @ 460.570309 Hz
undeformed
deformed
constraint
x
y
z

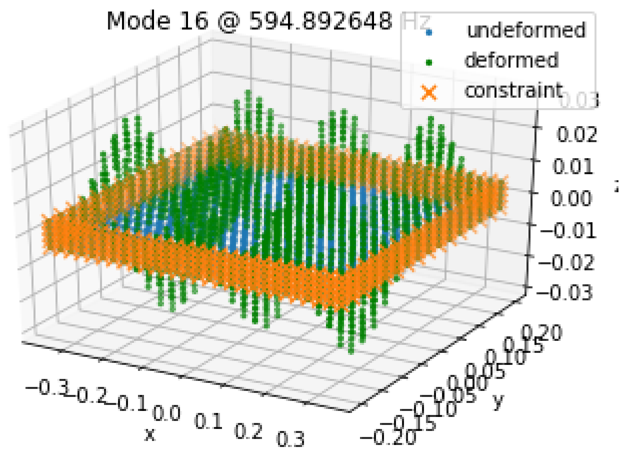

Mode 11 @ 469.752586 Hz
undeformed
deformed
constraint
x
y
z


Mode 12 @ 481.446785 Hz
undeformed
deformed
constraint
x
y
z


Mode 13 @ 507.539465 Hz
undeformed
deformed
constraint
x
y
z


Mode 14 @ 531.219220 Hz
undeformed
deformed
constraint
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
z
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
y


Mode 15 @ 555.617092 Hz
undeformed
deformed
constraint
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
z
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
y


Mode 16 @ 594.892648 Hz
undeformed
deformed
constraint
0.03
0.02
0.01
0.00
−0.01
−0.02
−0.03
z
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
x
−0.20
−0.15
−0.10
−0.05
0.00
0.05
0.10
0.15
0.20
y