2023-06-19_003_Graphs_Stress

June 19, 2023

[1]:
```
import numpy as np
import scipy
import imageio

import matplotlib
import matplotlib.pyplot as plt
import matplotlib.cm as cm

matplotlib.rc('image', interpolation='nearest')
matplotlib.rc('figure',facecolor='white')
matplotlib.rc('image',cmap='viridis')
colors=plt.rcParams['axes.prop_cycle'].by_key()['color']
%matplotlib inline

from matplotlib.animation import FuncAnimation
matplotlib.rc('animation',html='html5')
import colorcet as ccm
from graphplot import *
```

1 Example: Stress in a bridge

1.1 Building basic grid graph (representing a "bridge")

[2]:
```
def getPoslistNCube(shape,dtype=np.double):
    """Create list of positions in an n-dimensional cuboid of size shape."""
    ndim=len(shape)

    axGrids=[np.arange(i,dtype=dtype) for i in shape]
    prePos=np.array(np.meshgrid(*axGrids,indexing='ij'),dtype=dtype )
    # the first dimension of prepos is the dimension of the  posvector, the␣
↪successive dimensions are in the cube
    # so need to move first axis to end, and then flatten
    pos=np.rollaxis(prePos,0,ndim+1)
    # flattening
    newshape=(-1,ndim)
```

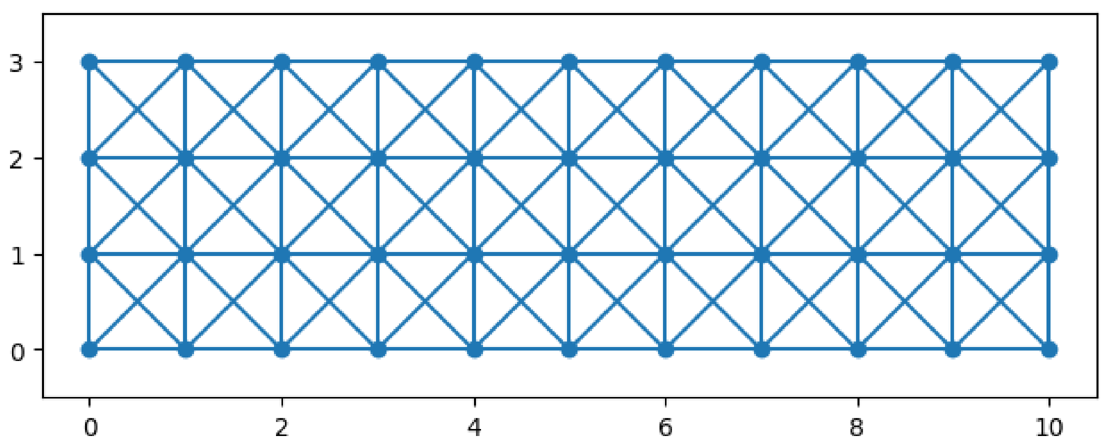
```
    return (pos.reshape(newshape)).copy()
```

[3]:

```
# build "rectangular" graphs
# how many rows and cols?
nRows=4
nCols=11
# spatial dimension
dim=2

# first create the vertex set
nList=[nCols,nRows]
posList=getPoslistNCube((nCols,nRows),dtype=np.int32)
nPoints=posList.shape[0]


# now add edges: we want to add edges according to an eight-neighbourhood
# the following gives the relative positions of all desired neighbours
# note: we only need "half" of them, otherwise each edge would be added twice
if False:
    # four neighbourhood
    relEdgeList=np.array([[1,0],[0,1]],dtype=np.int32)
if True:
    # eight neighbourhood
    relEdgeList=np.array([[1,0],[0,1],[1,1],[-1,1]],dtype=np.int32)

nRelEdges=relEdgeList.shape[0]

# add rel edges to all points, to get list of potential targets of edges
edgeTargets=posList.reshape((nPoints,1,dim))+relEdgeList.
 ↪reshape((1,nRelEdges,dim))
edgeInits=np.zeros((nPoints,nRelEdges,dim),dtype=np.int32)
for i in range(nRelEdges):
    edgeInits[:,i,:]=posList


# now identify invalid edges
keep=np.full((nPoints,nRelEdges),True)
for i in range(dim):
    keep*=(edgeTargets[:,:,i]<nList[i])
    keep*=(edgeTargets[:,:,i]>=0)

# now flatten edgeTarget and Init lists, then restrict to those we want to keep
edgeTargets=edgeTargets.reshape((-1,dim))
edgeInits=edgeInits.reshape((-1,dim))
keep=keep.ravel()
edgeTargets=edgeTargets[keep]
edgeInits=edgeInits[keep]
```

```python
# now convert coordinates to edge indices
idxStrides=np.array([nRows,1],dtype=np.int32)
edgeTargets=np.einsum(edgeTargets,[0,1],idxStrides,[1],[0])
edgeInits=np.einsum(edgeInits,[0,1],idxStrides,[1],[0])

nEdges=edgeTargets.shape[0]
edgeData=np.zeros((nEdges,2),dtype=np.int32)
edgeData[:,0]=edgeInits
edgeData[:,1]=edgeTargets

edgeLengths=np.linalg.norm(posList[edgeData[:,0]]-posList[edgeData[:,1]],axis=1)

# add reversed edges
#edgeTargets,edgeInits=np.concatenate((edgeTargets,edgeInits)),np.
 ↪concatenate((edgeInits,edgeTargets))
```

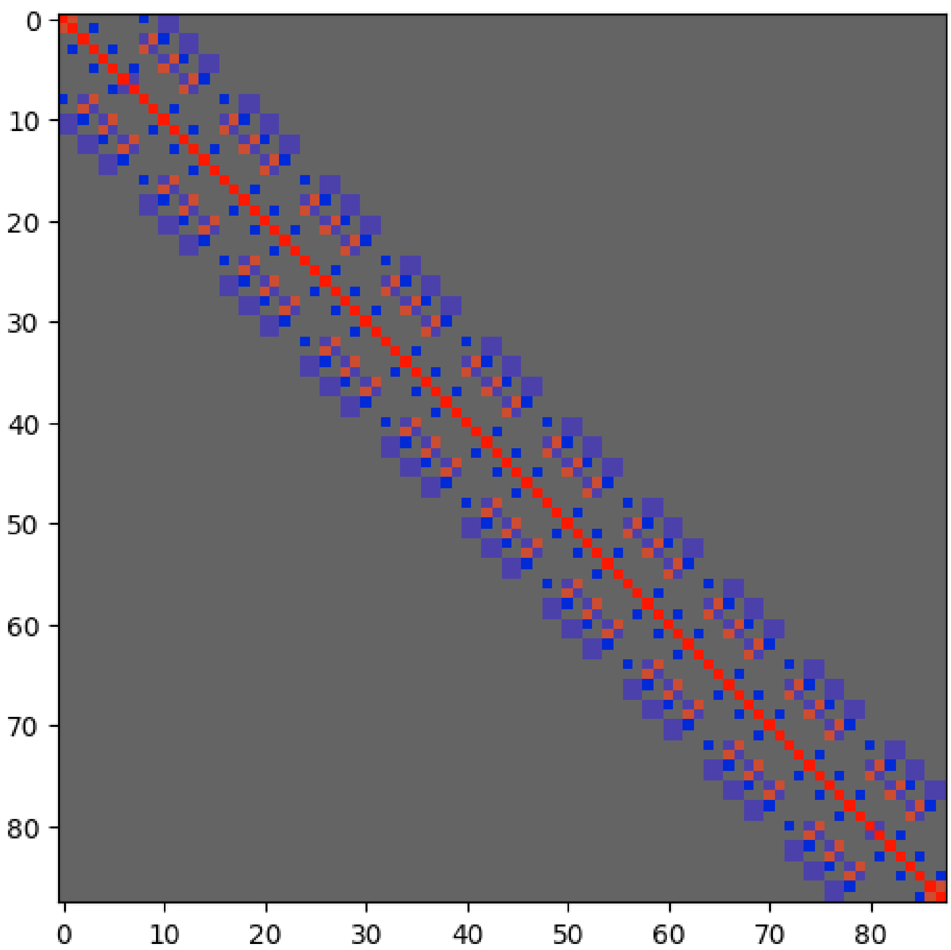
[4]:
```python
# basic display of graph: vertices and edges
fig=plt.figure()
ax=fig.add_subplot(aspect=1.)
vertices(ax,posList)
edges(ax,posList,edgeData)
setBoxLimits(ax,posList,buffer=0.5)
plt.tight_layout()
plt.show()
```



1.2 Graph Laplacian

[5]:
```
# build a particular version of the graph Laplacian
# that encodes first order deformation forces
# see simulation-forces-in-graph.pdf for some details

L=np.zeros((nPoints,dim,nPoints,dim),dtype=np.double)
for i in range(dim):
    for j in range(dim):
        L[edgeData[:,0],i,edgeData[:,1],j]=-(posList[edgeData[:,0],i]-\
                posList[edgeData[:,1],i])*\
                (posList[edgeData[:,0],j]-posList[edgeData[:,1],j])/
 ↪edgeLengths**2
# add reversed contributions
L+=L.transpose((2,1,0,3))

# diagonal entries
rng=np.arange(nPoints)
L[rng,:,rng,:]=-np.sum(L,axis=2)

# flatten
L=L.reshape((nPoints*dim,nPoints*dim))
```

[6]:
```
fig=plt.figure(figsize=(6,6))
vmax=0.2*np.max(np.abs(L))
plt.imshow(L,cmap=ccm.cm.CET_D8,vmin=-vmax,vmax=vmax)
plt.show()
```

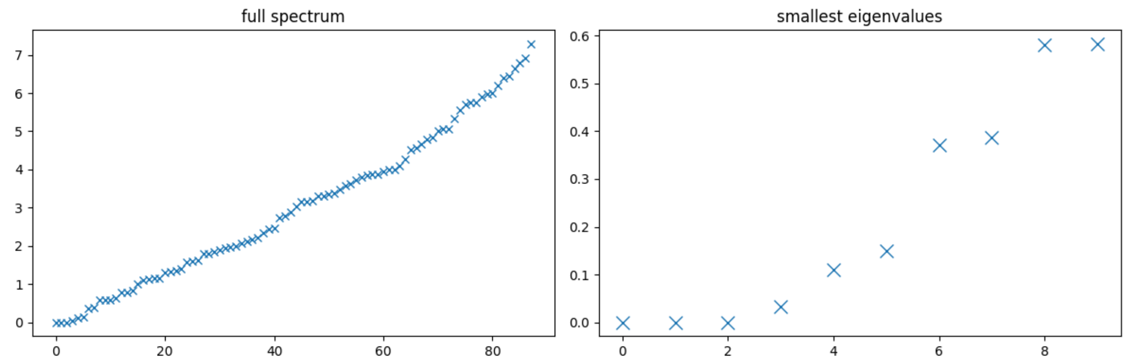


1.2.1 Eigenbasis of graph Laplacian

[7]:
```
eigData=np.linalg.eigh(L)
# some reordering:
# eigval: eigenvalues from small to large (more appropriate here)
# eigvec: eigenvectors for eigenvalues are in rows
eigval=eigData[0][::]
eigvec=(eigData[1].transpose())[::]
```

[8]:
```
# spectrum of Laplacian
fig=plt.figure(figsize=(12,4))
fig.add_subplot(1,2,1)
plt.title("full spectrum")
plt.plot(eigval,lw=0,marker="x")

fig.add_subplot(1,2,2)
```

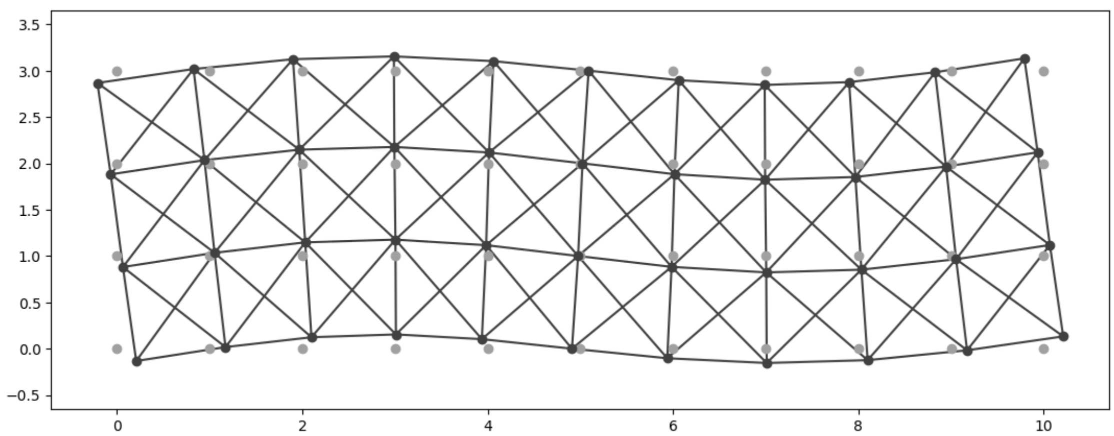
```
plt.title("smallest eigenvalues")
plt.plot(eigval[:10],lw=0,marker="x",markersize=10)

plt.tight_layout()
plt.show()
```



[9]:
```
# note: first three eigenvalues are zero.
# these correspond to "deformations"
# that do not change any pairwise lenghts
```

1.2.2 Visualize eigenmodes

[11]:
```
# static plot of "deformed graph"
i=5
mode=1.0*eigvec[i].reshape((-1,dim))
# visualize eigenmodes
fig=plt.figure(figsize=(12,12*(nRows/nCols)))
ax=fig.add_subplot(aspect=1.)
vertices(ax,posList,color="#a0a0a0")
vertices(ax,posList+mode,color="#404040")
edges(ax,posList+mode,edgeData,color="#404040")
setBoxLimits(ax,posList+mode,buffer=0.5)
plt.tight_layout()
plt.show()
```



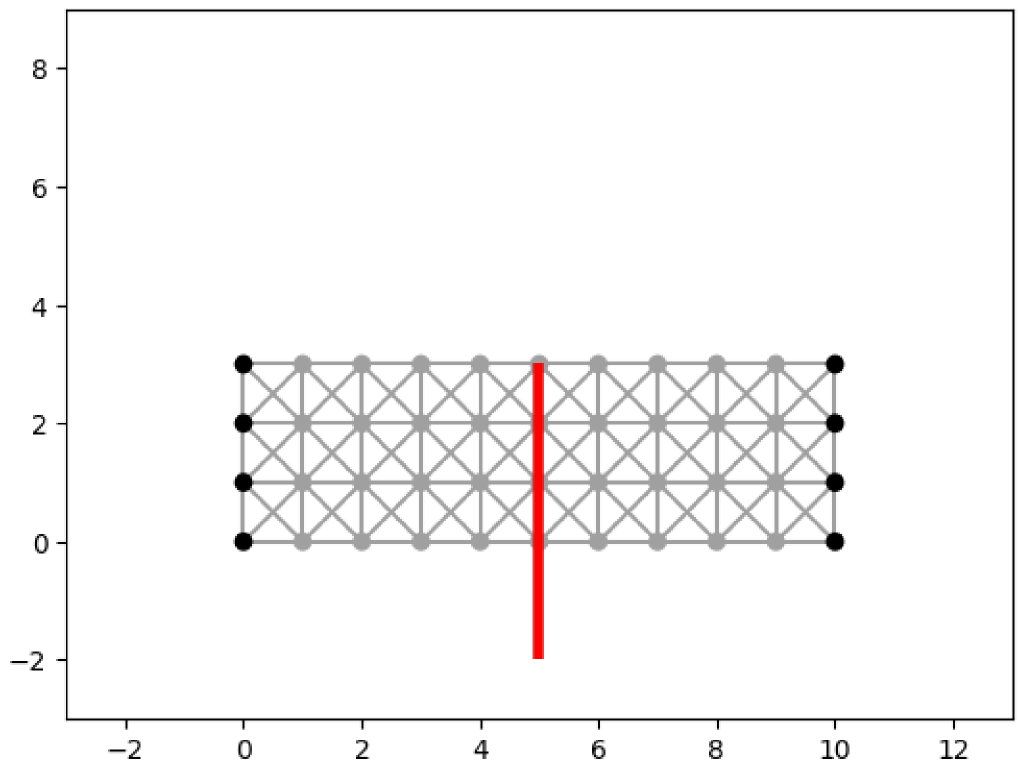
[12]:
```
# animation as oscillation
# will explain matplotlib animation later in more detail
i=3
mode=0.6*eigvec[i].reshape((-1,dim))

fig=matplotlib.figure.Figure(figsize=(12,4))
ax=fig.add_subplot(aspect=1.)

pltobj_pts = ax.scatter([], [],color="k")
pltobj_lineCollection=matplotlib.collections.
 ↪LineCollection([],zorder=-1,color="k")
ax.add_collection(pltobj_lineCollection)
ax.set_xlim([-0.5,nCols-0.5])
ax.set_ylim([-0.5,nRows-0.5])


def update(frame):
    data=posList+np.sin(frame)*mode
    pltobj_pts.set_offsets(data)
    pltobj_lineCollection.set_paths(data[edgeData])
    return pltobj_pts,pltobj_lineCollection,

ani = FuncAnimation(fig, update, frames=np.linspace(0, 2*np.pi, 20),
        blit=True,interval=1000/20)

matplotlib.rc('animation',html='jshtml')
ani
```

[12]: <matplotlib.animation.FuncAnimation at 0x7f17229618a0>

1.2.3 Application of external force, while keeping some vertices fixed

[13]:
```
V=np.zeros((nPoints,dim),dtype=np.double)
# apply a downward force on the upper middle point
V[nCols//2*nRows+nRows-1,1]=-1.
# keep vertices in first and last column fixed ("end points of bridge")
freeVertices=np.ones((nCols,nRows),dtype=bool)
freeVertices[0,:]=False
freeVertices[-1,:]=False
freeVertices=freeVertices.ravel()
nPointsFree=np.sum(freeVertices)
```

[14]:
```
mode=5.*V
# visualize forces
fig=plt.figure()
ax=fig.add_subplot(aspect=1.)
colList=["k" for _ in range(nPoints)]
for i in range(nPoints):
    if freeVertices[i]==True:
        colList[i]="#a0a0a0"
vertices(ax,posList,color=colList)

lineData=np.stack((posList,posList+mode),axis=1)
# simple plot as line collection
lc=matplotlib.collections.LineCollection(lineData,zorder=2,color="r",lw=4)
ax.add_collection(lc)


edges(ax,posList,edgeData,color="#a0a0a0")
plt.xlim([-3,nCols-1+3])
plt.ylim([-3,nRows-1+6])

plt.show()
```

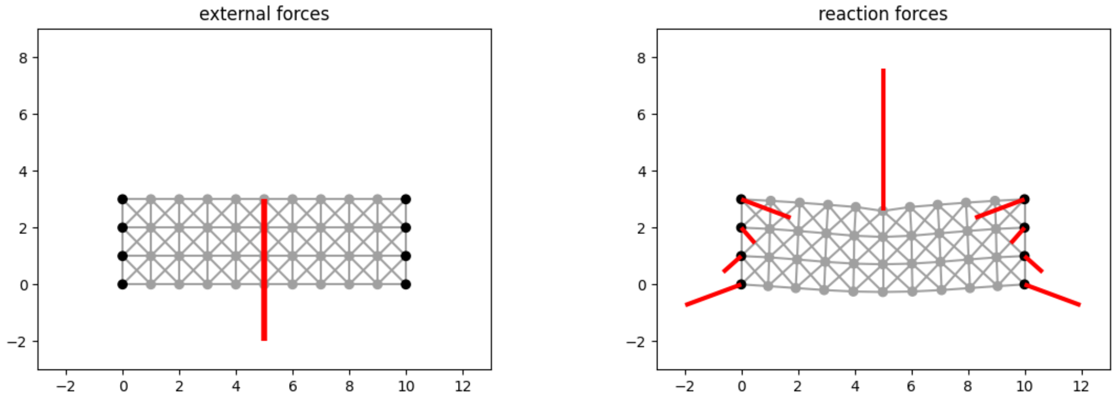


[15]:
```
VHat=V[freeVertices]
LHat=L.reshape((nPoints,dim,nPoints,dim))
LHat=LHat[freeVertices]
LHat=LHat[:,:,freeVertices]
LHat=LHat.reshape((nPointsFree*dim,nPointsFree*dim))

deformationData=np.linalg.lstsq(LHat,VHat.ravel(),rcond=None)

deformationHat=deformationData[0]
deformation=np.zeros((nPoints,dim),dtype=np.double)
deformation[freeVertices,:]=deformationHat.reshape((nPointsFree,dim))
```

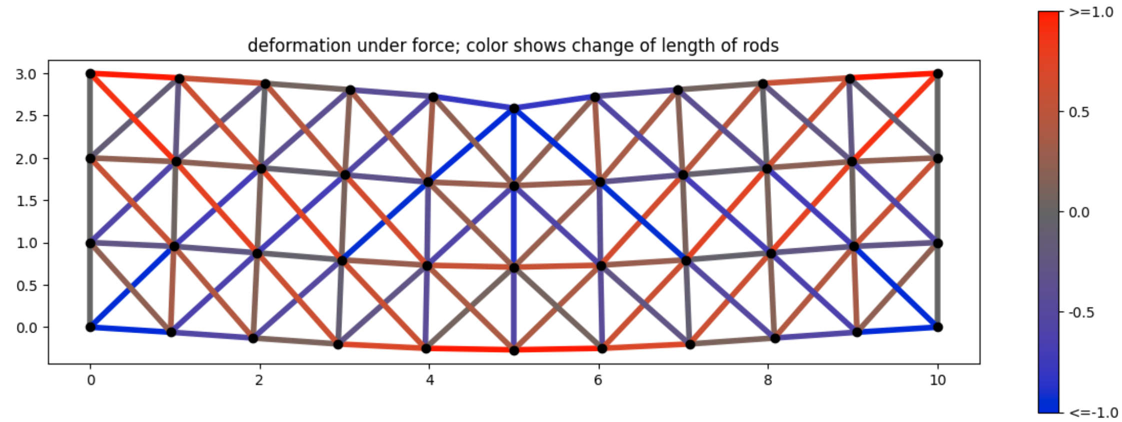
[16]:
```
# what are the resulting forces of the bridge deformation
force=-L.dot(deformation.ravel()).reshape((-1,dim))
```

[17]:
```
# visualize external and reaction forces
fig=plt.figure(figsize=(12,4))


ax=fig.add_subplot(1,2,1,aspect=1.)
mode=5.*V
plt.title("external forces")
```

```
colList=["k" for _ in range(nPoints)]
for i in range(nPoints):
    if freeVertices[i]==True:
        colList[i]="#a0a0a0"
vertices(ax,posList,color=colList)

lineData=np.stack((posList,posList+mode),axis=1)
# simple plot as line collection
lc=matplotlib.collections.LineCollection(lineData,zorder=2,color="r",lw=4)
ax.add_collection(lc)


edges(ax,posList,edgeData,color="#a0a0a0")
plt.xlim([-3,nCols-1+3])
plt.ylim([-3,nRows-1+6])


ax=fig.add_subplot(1,2,2,aspect=1.)
modeForce=5.*force
mode=0.2*deformation
plt.title("reaction forces")
colList=["k" for _ in range(nPoints)]
for i in range(nPoints):
    if freeVertices[i]==True:
        colList[i]="#a0a0a0"
vertices(ax,posList+mode,color=colList)

lineData=np.stack((posList+mode,posList+mode+modeForce),axis=1)
lc=matplotlib.collections.LineCollection(lineData,zorder=1,color="r",lw=3)
ax.add_collection(lc)

edges(ax,posList+mode,edgeData,color="#a0a0a0")
plt.xlim([-3,nCols-1+3])
plt.ylim([-3,nRows-1+6])


plt.tight_layout()
plt.show()
```



[18]:
```
mode=0.2*deformation.reshape((-1,dim))
posDef=posList+mode
edgeLengthsNew=np.linalg.norm(posDef[edgeData[:,0]]-posDef[edgeData[:
↪,1]],axis=1)
edgeLengthsDelta=edgeLengthsNew-edgeLengths

vmax=0.5*np.max(np.abs(edgeLengthsDelta))
colfun=ccm.cm.CET_D8
edgeColors=colfun(np.clip((1+edgeLengthsDelta/vmax)/2,0,1))

fig=plt.figure(figsize=(12,12*nRows/nCols))
ax=fig.add_subplot(aspect=1.)
vertices(ax,posList+mode,c="k")
edges(ax,posList+mode,edgeData,colors=edgeColors,lw=4)
ticks=np.linspace(-1,1,5)
cbar=fig.colorbar(
        cm.ScalarMappable(norm=matplotlib.colors.Normalize(vmin=-1, vmax=1,␣
↪clip=False), cmap=colfun),
        ax=ax, ticks=ticks)
# manually change labels of colorbar ticks
ticksLabels=[str(i) for i in ticks]
ticksLabels[0]="<="+ticksLabels[0]
ticksLabels[-1]=">="+ticksLabels[-1]
cbar.ax.set_yticklabels(ticksLabels)


plt.title("deformation under force; color shows change of length of rods")
plt.tight_layout()
plt.show()
```



[19]:
```
mode=0.1*deformation.reshape((-1,dim))

# try to estimate maximal deformation
data=posList+mode
edgeLengthsNew=np.linalg.norm(data[edgeData[:,0]]-data[edgeData[:,1]],axis=1)
edgeLengthsDelta=edgeLengthsNew-edgeLengths
vmax=0.5*np.max(np.abs(edgeLengthsDelta))
colfun=ccm.cm.CET_D8


fig=matplotlib.figure.Figure(figsize=(12,4))
ax=fig.add_subplot(aspect=1.)

pltobj_pts = ax.scatter([], [],c="k")
#lineCollection=matplotlib.collections.
 ↪LineCollection(posList[edgeData],zorder=-1)
pltobj_lineCollection=matplotlib.collections.
 ↪LineCollection([[[0,0],[1,1]]],zorder=-1,lw=4)
ax.add_collection(pltobj_lineCollection)
ax.set_xlim([-0.5,nCols-0.5])
ax.set_ylim([-0.5,nRows-0.5])

#def init():
#    ax.set_xlim([-0.5,nCols-0.5])
#    ax.set_ylim([-0.5,nRows-0.5])
#    return ln,

def update(frame):
    data=posList+np.sin(frame)**2*mode

    edgeLengthsNew=np.linalg.norm(data[edgeData[:,0]]-data[edgeData[:
 ↪,1]],axis=1)
```

```
    edgeLengthsDelta=edgeLengthsNew-edgeLengths

    edgeColors=colfun(np.clip((1+edgeLengthsDelta/vmax)/2,0,1))


    pltobj_pts.set_offsets(data)
    pltobj_lineCollection.set_paths(data[edgeData])
    pltobj_lineCollection.set_color(edgeColors)
    return pltobj_pts,pltobj_lineCollection,
    #ln2, = plt.plot([], [], 'ro')
    #return ln2,
#ani = FuncAnimation(fig, update, frames=np.linspace(0, 2*np.pi, 20),
#                     init_func=init, blit=True,interval=1000/20)
ani = FuncAnimation(fig, update, frames=np.linspace(0, np.pi, 20),
        blit=True,interval=1000/20*2)

matplotlib.rc('animation',html='jshtml')
ani
```

[19]: <matplotlib.animation.FuncAnimation at 0x7f1720cdbb50>

[ ]: