2023-06-19_010_Meshes_Part-02

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 *
```

[2]:
```
import FEM as FEM
```

1 Meshes: Regions and boundaries

- intution: every surface has a boundary, which is a contour
- every path has a boundary, which consists of start and endpoint
- closed surfaces and cycles have an ”empty” boundary
- this is the beginning of a whole branch of pure and abstract mathematics
- but now and then these questions arise in our ”everyday research”, then it is helpful to have heard about these ideas

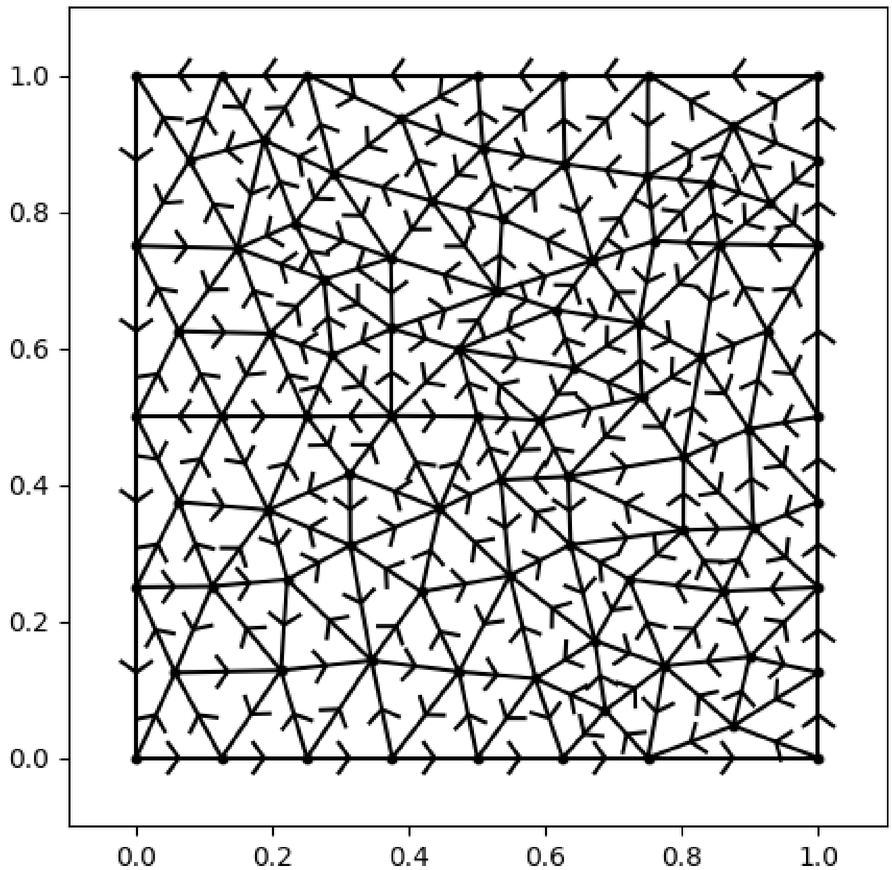
1.1 Setup of a simple example mesh

[3]:
```
# again: create/load a finer example mesh
data=np.load("triangulation.npz")
pointData=data["pointData"]
triangleData=data["triangleData"]
data.close()
nPoints=pointData.shape[0]
nTriangles=triangleData.shape[0]

# extract edge data
edgeData,etAdjacencyData,etAdjacencyDataOrientation=FEM.getEdges(triangleData)
nEdges=edgeData.shape[0]
```

[4]:
```
def␣
↪drawEdges(ax,pointData,edgeData,colors="k",drawOrientations=False,arrowScale=0.
↪03,**kwargs):
    lineCollection=matplotlib.collections.
↪LineCollection(pointData[edgeData],color=colors,**kwargs)
    ax.add_collection(lineCollection)

    # draw small (manual) on edges
    if drawOrientations:
        edgeCoords=pointData[edgeData]
        midPoints=np.mean(edgeCoords,axis=1)
        orientations=edgeCoords[:,1,:]-edgeCoords[:,0,:]
        norms=np.linalg.norm(orientations,axis=1)
        orientations=np.einsum(orientations,[0,1],1./norms,[0],[0,1])
        for phi in [0.7*np.pi,-0.7*np.pi]:
            R=np.array([[np.cos(phi),-np.sin(phi)],[np.sin(phi),np.
↪cos(phi)]],dtype=np.double)
            arr=np.einsum(R,[0,1],orientations,[2,1],[2,0])
            lines=np.zeros((nEdges,2,2))
            lines[:,0,:]=midPoints
            lines[:,1,:]=midPoints+arrowScale*arr
            lineCollection=matplotlib.collections.
↪LineCollection(lines,color=colors,**kwargs)
            ax.add_collection(lineCollection)
```

[5]:
```
fig=plt.figure()
ax=fig.add_subplot(aspect=1.)
ax.scatter(pointData[:,0],pointData[:,1],c="k",s=10)

drawEdges(ax,pointData,edgeData,colors="k",drawOrientations=True,arrowScale=0.
↪03)
#ax.scatter(midPoints[:,0],midPoints[:,1])
```

```
buffer=0.1
plt.xlim([-buffer,1+buffer])
plt.ylim([-buffer,1+buffer])
plt.tight_layout()
plt.show()
```



1.2 Representing regions

- now we can represent 2d regions that are given as unions of triangles
- we simply use an indicator vector: 1 if triangle is part of region, 0 otherwise

1.2.1 Example: single triangle

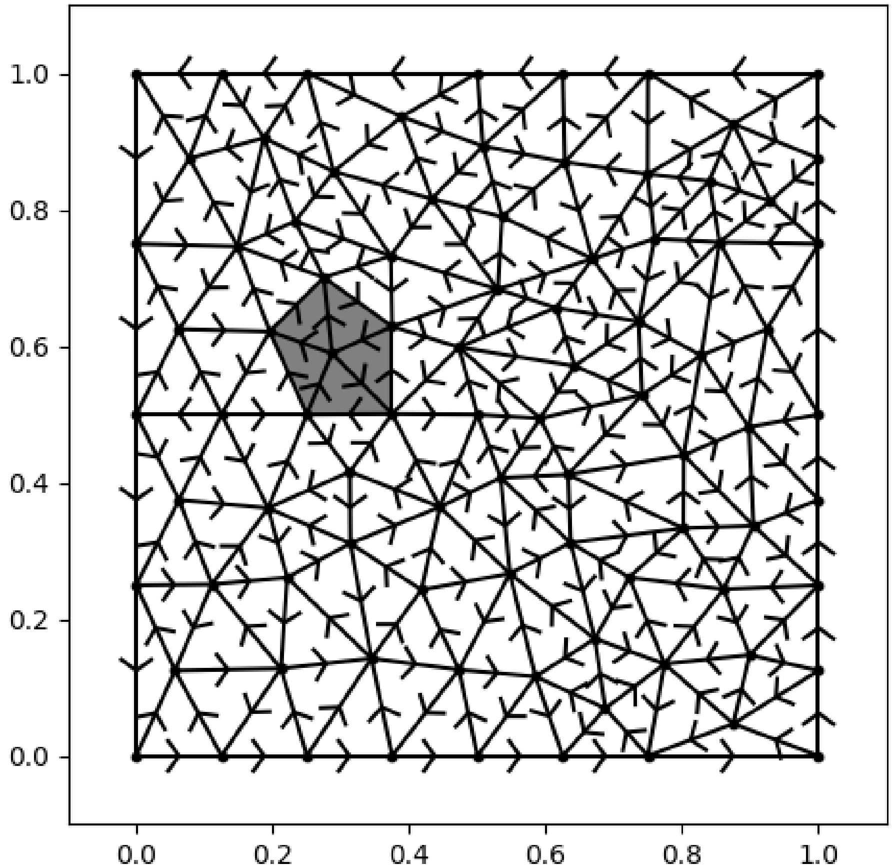
[6]:
```
# allocate an empty region, then set some index to True
region=np.zeros((nTriangles,),dtype=bool)
region[100]=True
```

1.2.2 Example: a few more triangles

[9]:
```
# allocate an empty region, then set some index to True
# find all triangles that touch a given vertex
region=(np.sum(triangleData==20,axis=1)>0)
```

1.3 Visualization of region

[10]:
```
# now visualize this as filled triangles
fig=plt.figure()
ax=fig.add_subplot(aspect=1.)
ax.scatter(pointData[:,0],pointData[:,1],c="k",s=10)

drawEdges(ax,pointData,edgeData,colors="k",drawOrientations=True)

polyCollection=matplotlib.collections.
 ↪LineCollection(pointData[triangleData[region]],lw=0,fc="#808080",zorder=-2)
ax.add_collection(polyCollection)

buffer=0.1
plt.xlim([-buffer,1+buffer])
plt.ylim([-buffer,1+buffer])
plt.tight_layout()
plt.show()
```



1.4 Compute boundary of region

[12]:
```
def getB2(nEdges,etAdjacencyData,etAdjacencyDataOrientation):
    nTriangles=etAdjacencyData.shape[0]
    data=etAdjacencyDataOrientation.copy().ravel()
    indices=etAdjacencyData.copy().ravel()
    indptr=3*np.arange(0,nTriangles+1)
    result=scipy.sparse.
 ↪csc_matrix((data,indices,indptr),shape=(nEdges,nTriangles)).tocsr()
    return result
```

[13]:
```
# boundary operator: take indicator vector of triangles to indicator vector of␣
 ↪bounding edges
# take orientation into account: this way, contributions from adjacent␣
 ↪triangles cancel

# B2 stands for: boundary of 2-dimensional objects
```

```
B2=getB2(nEdges,etAdjacencyData,etAdjacencyDataOrientation)
```

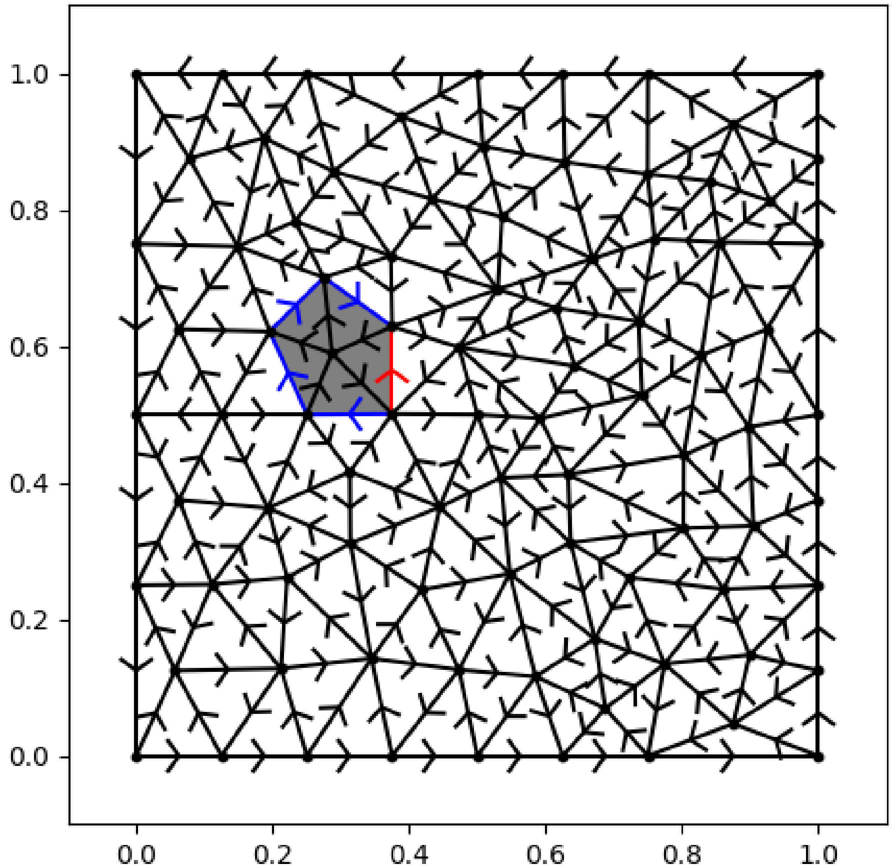
[14]:
```
B2.shape
```

[14]: (237, 150)

[15]:
```
# now apply to region indicator
boundary=B2.dot(region)
```

[16]:
```
def signToColor(bdry):
    colors=np.zeros((bdry.shape[0],3))
    colors[bdry>0.1]=np.array([1.,0.,0.])
    colors[bdry<-0.1]=np.array([0.,0.,1.])
    return colors
```

[17]:
```
# highlight boundary in plot
fig=plt.figure()
ax=fig.add_subplot(aspect=1.)
ax.scatter(pointData[:,0],pointData[:,1],c="k",s=10)

lineColors=signToColor(boundary)
drawEdges(ax,pointData,edgeData,colors=lineColors,drawOrientations=True,arrowScale=0.
 ↪03,zorder=-1)

polyCollection=matplotlib.collections.
 ↪LineCollection(pointData[triangleData[region]],lw=0,fc="#808080",zorder=-2)
ax.add_collection(polyCollection)

buffer=0.1
plt.xlim([-buffer,1+buffer])
plt.ylim([-buffer,1+buffer])
plt.tight_layout()
plt.show()
```



1.5 Shortest paths

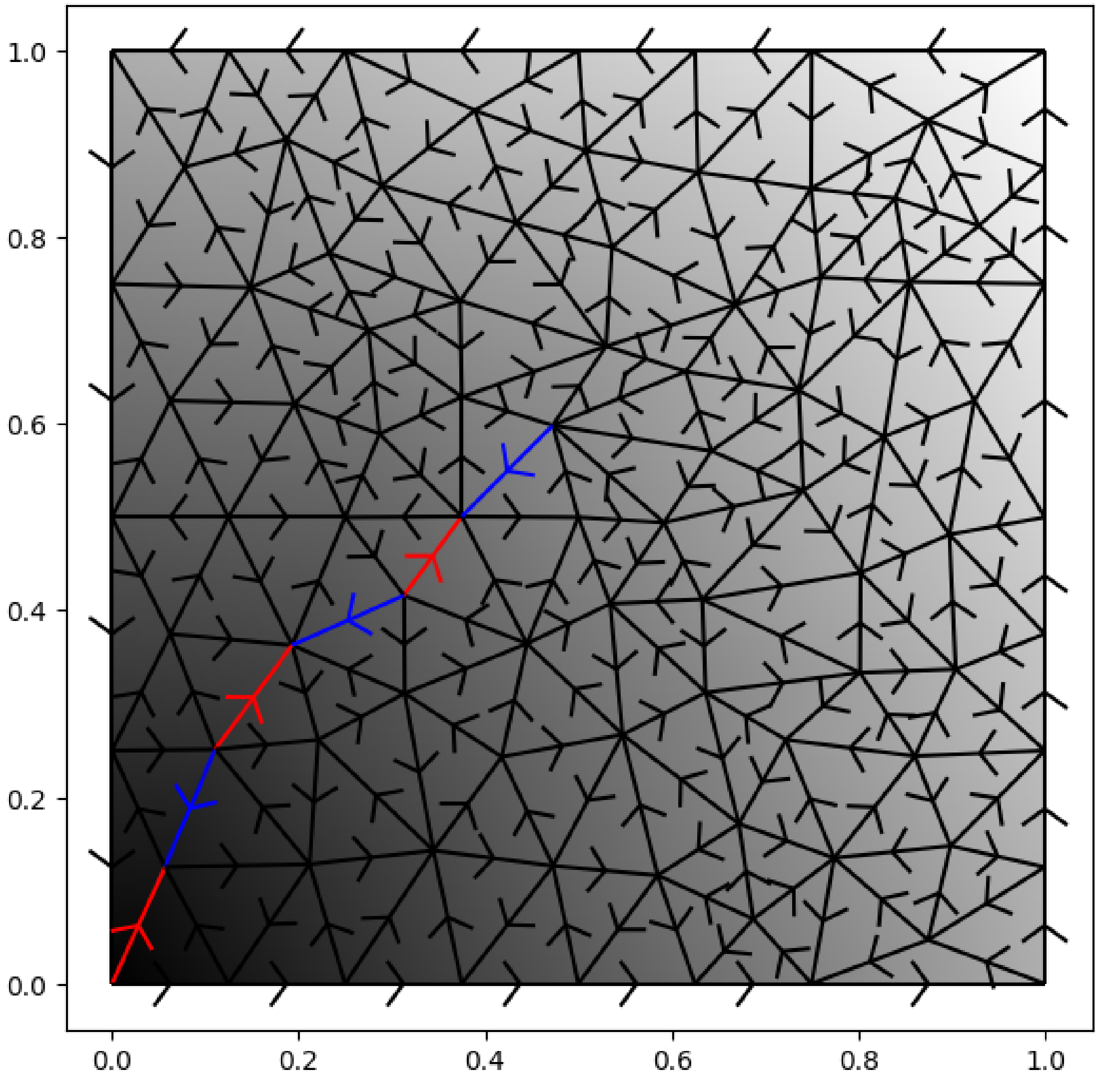
We can use Dijkstra's algorithm (or others) to find shortest paths on edges through the mesh

[18]:
```
def getShortestPaths(pointData,edgeData):
    # compute edge lengths
    edgeLengths=np.linalg.norm(pointData[edgeData[:,0]]-pointData[edgeData[:
↪,1]],axis=1)
    # build weighted adjacency matrix of graph, with weights given by lengths

    if False:
        # note: edges in the graph that we want to perform the search on, are␣
↪un-oriented, but the computational
        # representation has oriented edges. so we need to add both directions
        # the construction is most efficiently done in scipy.sparse.coo_matrix:
        graph=scipy.sparse.coo_matrix((\
                np.concatenate((edgeLengths,edgeLengths)),\
                (np.concatenate((edgeData[:,0],edgeData[:,1])),\
```

```
                np.concatenate((edgeData[:,1],edgeData[:,0])))\
                ))
        # non-existing entries correspond to infinite edge lengths (not zero!)
        # convert to csr, which is (probably) faster in graph algorithm
        graph=graph.tocsr()
    else:
        # use "directed=False" option of graph search instead
        graph=scipy.sparse.coo_matrix((edgeLengths,(edgeData[:,0],edgeData[:
↪,1])))


    dist,pred=scipy.sparse.csgraph.
↪shortest_path(graph,directed=False,return_predecessors=True)
    return dist,pred
```

[19]:
```
def extractPath(pred,i0,i1):
    path=[i1]
    j=i1
    while j!=i0:
        j=pred[i0,j]
        path.append(j)
    return np.array(path,dtype=np.int32)


def translatePath(path,edgeData):
    nEdges=edgeData.shape[0]
    # these matrices will help identifying edge indices from the vertex indices:
    auxmatInd=scipy.sparse.coo_matrix((\
            np.concatenate((np.arange(nEdges),np.arange(nEdges))),\
            (np.concatenate((edgeData[:,0],edgeData[:,1])),\
            np.concatenate((edgeData[:,1],edgeData[:,0])))\
            )).tocsr()
    auxmatOrientation=scipy.sparse.coo_matrix((\
            np.concatenate((np.full((nEdges,),fill_value=1.),np.
↪full((nEdges,),fill_value=-1.))),\
            (np.concatenate((edgeData[:,0],edgeData[:,1])),\
            np.concatenate((edgeData[:,1],edgeData[:,0])))\
            )).tocsr()

    result=np.zeros((nEdges,),dtype=np.double)
    for j in range(len(path)-1,0,-1):
        ␣
↪result[auxmatInd[path[j],path[j-1]]]=auxmatOrientation[path[j],path[j-1]]
    return result
```

[20]:
```python
# plot length from a given vertex as color coded mesh
# which vertex?
i0=0
# find shortest path to a given target vertex
i1=30

dist,pred=getShortestPaths(pointData,edgeData)
pathList=extractPath(pred,i0,i1)
path=translatePath(pathList,edgeData)


fig=plt.figure(figsize=(6,6))
ax=fig.add_subplot(aspect=1.)

tri=matplotlib.tri.Triangulation(pointData[:,0],pointData[:,1],triangleData)
pltobj=ax.tripcolor(tri,dist[i0,:]/np.max(dist[i0,:]),shading='gouraud',cmap=cm.
 ↪gray)

lineColors=signToColor(path)
drawEdges(ax,pointData,edgeData,colors=lineColors,drawOrientations=True,arrowScale=0.
 ↪03,zorder=1)

#plt.colorbar(pltobj)

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



1.6 Boundaries of paths

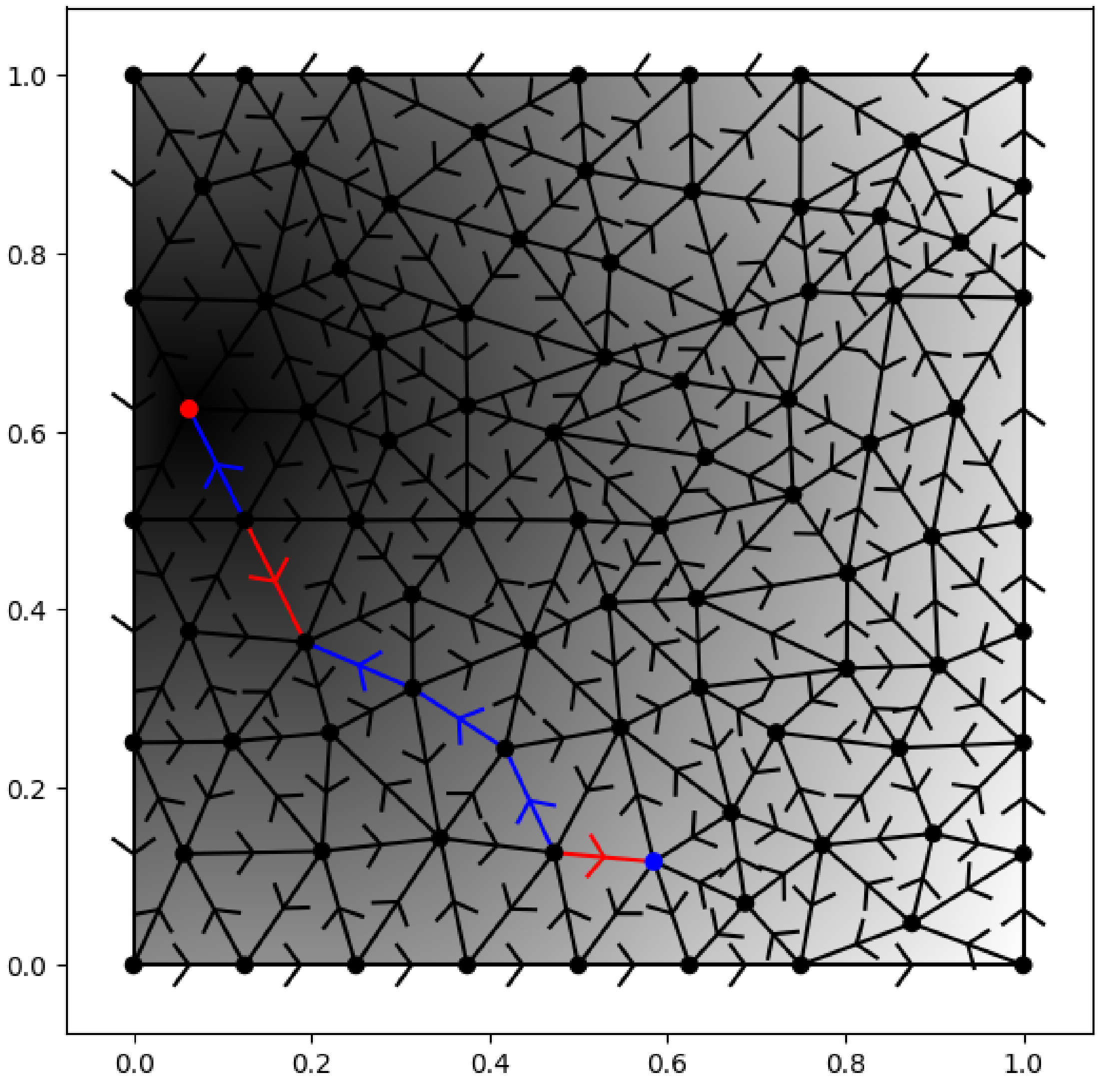
- We can interpret starting and end points of lines (paths) as their boundaries, weighted with opposite signs.
- A path from point A to point B will then have A as positive boundary and B as negative boundary.

[20]:
```
def getB1(nPoints,edgeData):
    nEdges=edgeData.shape[0]
    data=np.zeros((nEdges*2),dtype=np.double)
    data[0::2]=1.
    data[1::2]=-1.
```

```
    indices=edgeData.copy().ravel()
    indptr=2*np.arange(nEdges+1)
    result=scipy.sparse.
↪csc_matrix((data,indices,indptr),shape=(nPoints,nEdges)).tocsr()
    return result
```

[21]:
```
B1=getB1(nPoints,edgeData)
```

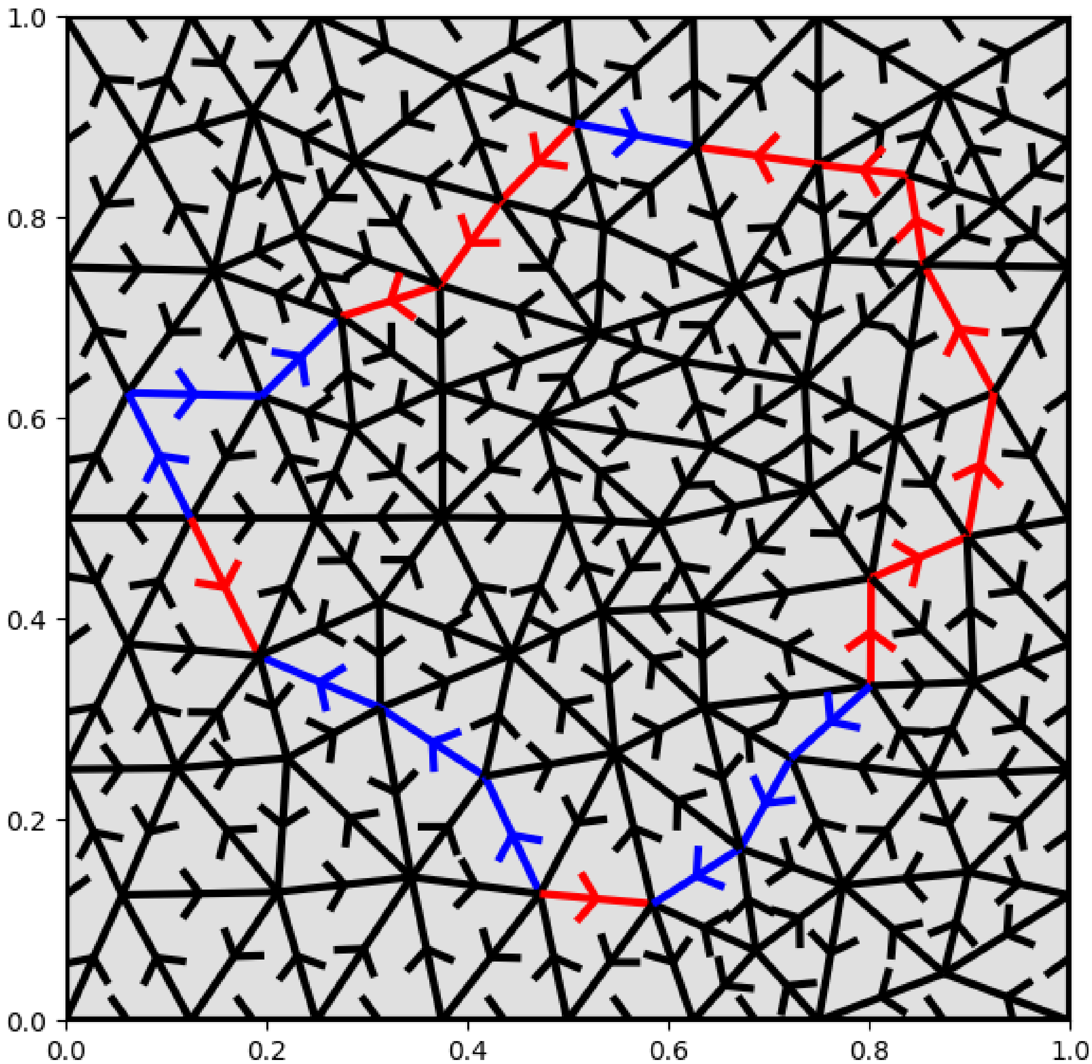
[22]:
```
# starting vertex
i0=10
# end vertex
i1=53

pathList=extractPath(pred,i0,i1)
path=translatePath(pathList,edgeData)



fig=plt.figure(figsize=(6,6))
ax=fig.add_subplot(aspect=1.)

tri=matplotlib.tri.Triangulation(pointData[:,0],pointData[:,1],triangleData)
ax.tripcolor(tri,dist[i0,:]/np.max(dist[i0,:]),shading='gouraud',cmap=cm.gray)

lineColors=signToColor(path)
drawEdges(ax,pointData,edgeData,colors=lineColors,drawOrientations=True,arrowScale=0.
↪03,zorder=1)

# add colored points to indicate boundary of shortest path
pathBoundary=B1.dot(path)
pointColors=signToColor(pathBoundary)
plt.scatter(pointData[:,0],pointData[:,1],c=pointColors,zorder=2)

#for i,x in enumerate(pointData):
#    ax.text(x[0],x[1],i,c="r")

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



1.7 Region from boundary

[23]:
```
# first build a cycle (which we conjecture to be a boundary)
# fix some vertices, connect shortest paths between them
iList=[10,53,31,56,10]
path=np.zeros(nEdges)
for j in range(len(iList)-1):
    pathList=extractPath(pred,iList[j],iList[j+1])
    path+=translatePath(pathList,edgeData)

fig=plt.figure(figsize=(6,6),facecolor="w")
ax=fig.add_subplot(aspect=1.,facecolor="#e0e0e0")
```

```
lineColors=signToColor(path)
drawEdges(ax,pointData,edgeData,colors=lineColors,drawOrientations=True,arrowScale=0.
 ↪03,zorder=1,lw=3)

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



[24]:
```
import scipy.sparse.linalg
```

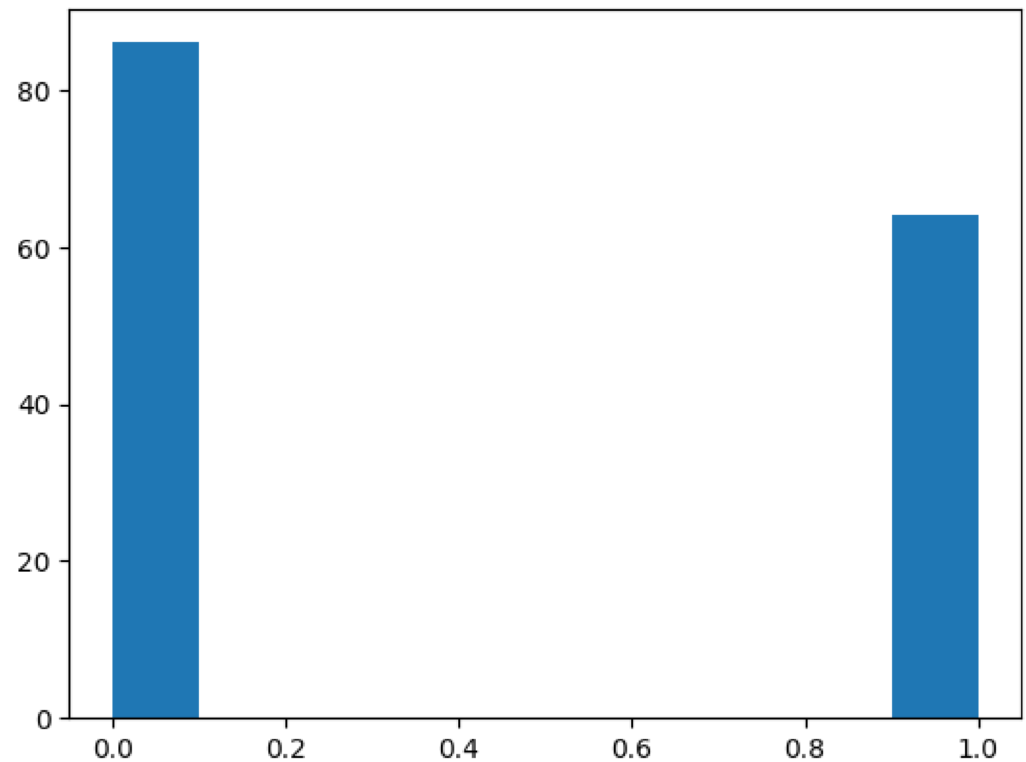
[25]:
```
# next: search for region indicator, such that it has the specified path as␣
 ↪boundary.
# solve: B2.dot(region)=path
```

```
# since we do not know for sure if solution exists, use leastsquares ansatz
res=scipy.sparse.linalg.lsqr(B2,path)
region=res[0]
# test solution quality:
print("norm of discrepancy between B2(region) and path:",res[3])
print("if solution was found, this should be 1: ",res[1])
```

```
norm of discrepancy between B2(region) and path: 8.613445229696281e-05
if solution was found, this should be 1:  1
```

[26]:

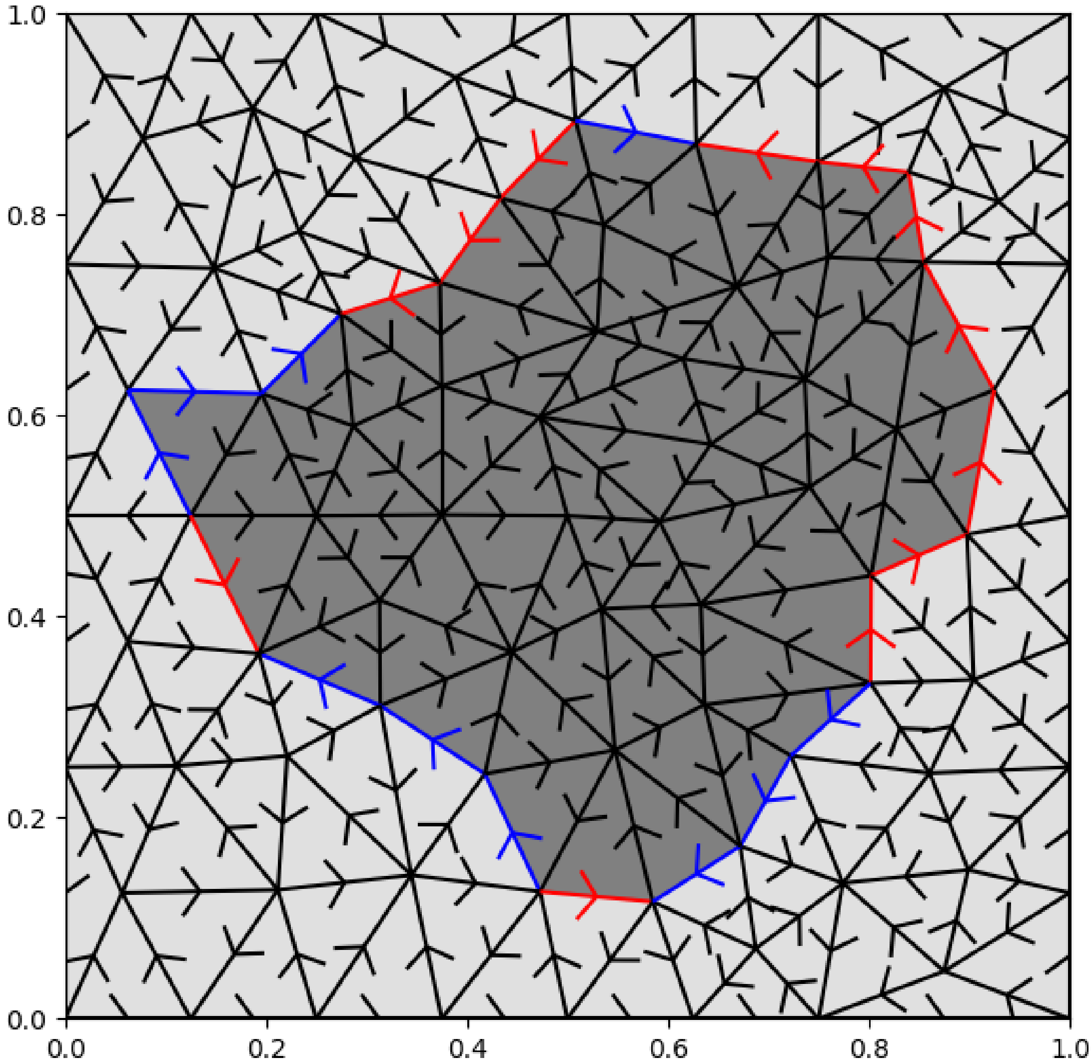
```
# if a good solution was found, then region should be binary, with only 0 and 1␣
↪entries
plt.hist(region)
plt.show()
```



[27]:

```
# first build a cycle (which we conjecture to be a boundary)
# fix three vertices, connect shortest paths between them

fig=plt.figure(figsize=(6,6),facecolor="w")
ax=fig.add_subplot(aspect=1.,facecolor="#e0e0e0")

lineColors=signToColor(path)
```

```
drawEdges(ax,pointData,edgeData,colors=lineColors,drawOrientations=True,arrowScale=0.
↪03,zorder=1)

regionIndicator=(region>0.1)
polyCollection=matplotlib.collections.
↪LineCollection(pointData[triangleData[regionIndicator]],\
        lw=0,fc="#808080",zorder=-2)
ax.add_collection(polyCollection)


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



1.8 Is every cycle path a boundary of a region?

[28]:
```
# first observation: each boundary path of a region has zero boundary itself
# (boundaries of regions are cycle paths; these have no beginning or end point)
# confirm by showing that B1.dot(B2)= zero matrix
B1.dot(B2)
```

[28]:
```
<88x150 sparse matrix of type '<class 'numpy.float64'>'
        with 0 stored elements in Compressed Sparse Row format>
```

[29]:
```
# so: image of B2 lives in kernel of B1
# compare dimensions:
#   dim(ker B1)=nEdges-rank(B1)
#   dim(img B2)=rank(B2)
```

[30]:
```
# since we use small matrices: determine rank on dense matrices
B1Dense=B1.todense()
B2Dense=B2.todense()

print(nEdges-np.linalg.matrix_rank(B1Dense))
print(np.linalg.matrix_rank(B2Dense))
```

```
150
150
```

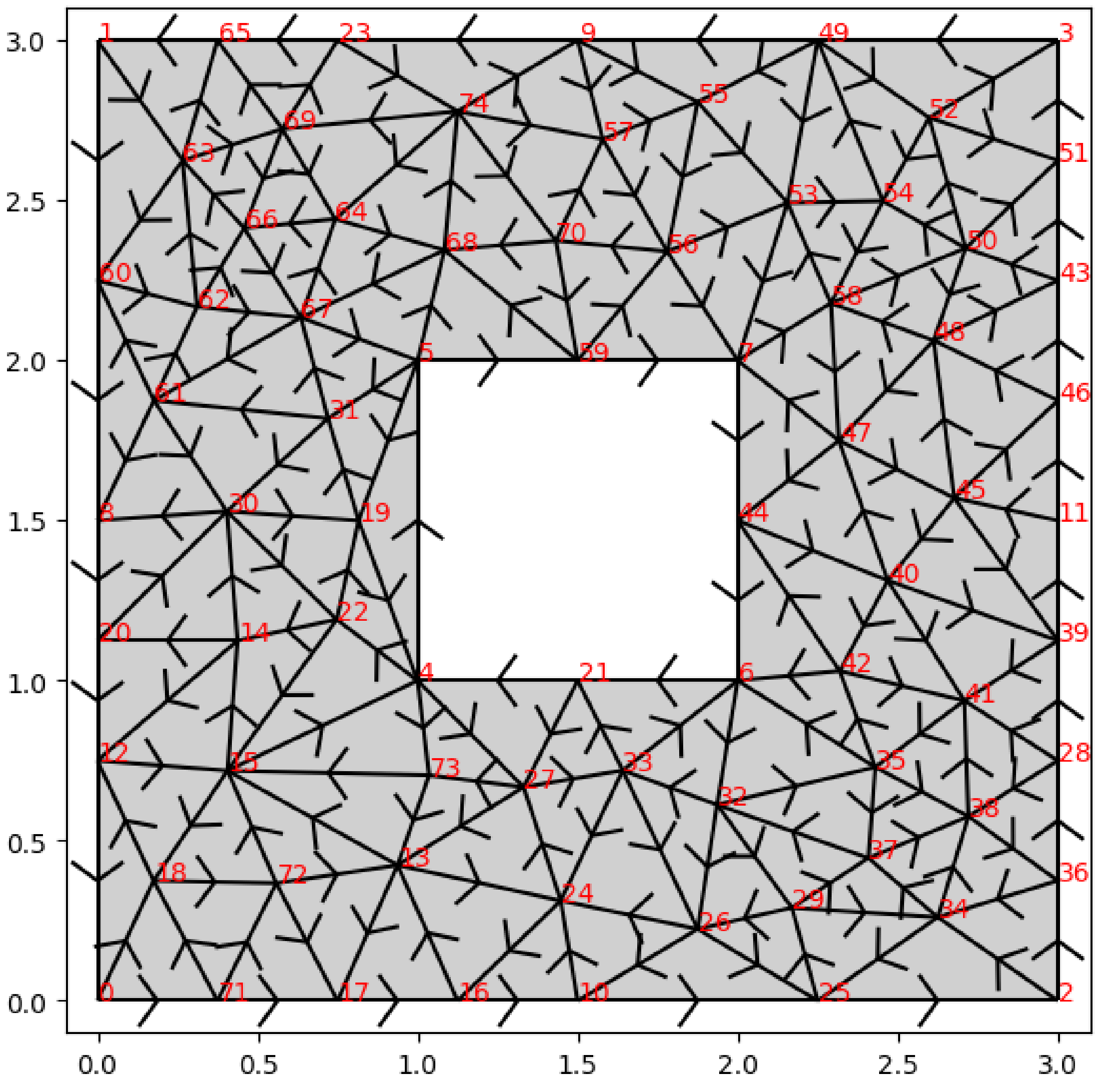
[31]:
```
# both numbers are equal: hence every cycle must be a boundary on our current␣
↪mesh
```

1.9 A mesh with a hole

[33]:
```
# create/load a finer example mesh
if False:
    import lib.Triangle as Triangle

    points=np.array([[0,0],[0,3],[3,0],[3,3],[1,1],[1,2],[2,1],[2,2]],dtype=np.
↪double)
    segments=np.array([[4,5],[5,7],[7,6],[6,4]],dtype=np.int32)
    segmentMarkers=np.zeros(segments.shape[0],dtype=np.int32)
    holes=np.array([[1.5,1.5]],dtype=np.double)
    maxArea=0.1
    pointData,triangleData=Triangle.
↪createMeshAdvanced(points,segments,segmentMarkers,holes,\
            flagStr="zqpc",maxArea=maxArea)
    np.savez("triangulation_hole.
↪npz",pointData=pointData,triangleData=triangleData)
else:
```

```
    data=np.load("triangulation_hole.npz")
    pointData=data["pointData"]
    triangleData=data["triangleData"]
    data.close()
nPoints=pointData.shape[0]
nTriangles=triangleData.shape[0]

# extract edge data
edgeData,etAdjacencyData,etAdjacencyDataOrientation=FEM.getEdges(triangleData)
nEdges=edgeData.shape[0]
```

[34]:
```
fig=plt.figure(figsize=(6,6),facecolor="w")
ax=fig.add_subplot(aspect=1.)

polyCollection=matplotlib.collections.
 ↪LineCollection(pointData[triangleData],lw=0,fc="#d0d0d0",zorder=-2)
ax.add_collection(polyCollection)
drawEdges(ax,pointData,edgeData,colors="k",drawOrientations=True,arrowScale=0.1)

for i,x in enumerate(pointData):
    ax.text(x[0],x[1],i,c="r")

# plot range
buffer=0.1
lim1,lim2,lim3,lim4=np.min(pointData[:,0]),np.max(pointData[:,0]),np.
 ↪min(pointData[:,1]),np.max(pointData[:,1])
plt.xlim([lim1-buffer,lim2+buffer])
plt.ylim([lim3-buffer,lim4+buffer])


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



[35]:
```
B1=getB1(nPoints,edgeData)
B2=getB2(nEdges,etAdjacencyData,etAdjacencyDataOrientation)
```

[36]:
```
# do the rank test again:

# recall:
# -image of B2 lives in kernel of B1
# -compare dimensions:
#   dim(ker B1)=nEdges-rank(B1)
#   dim(img B2)=rank(B2)

# since we use small matrices: determine rank on dense matrices
B1Dense=B1.todense()
```

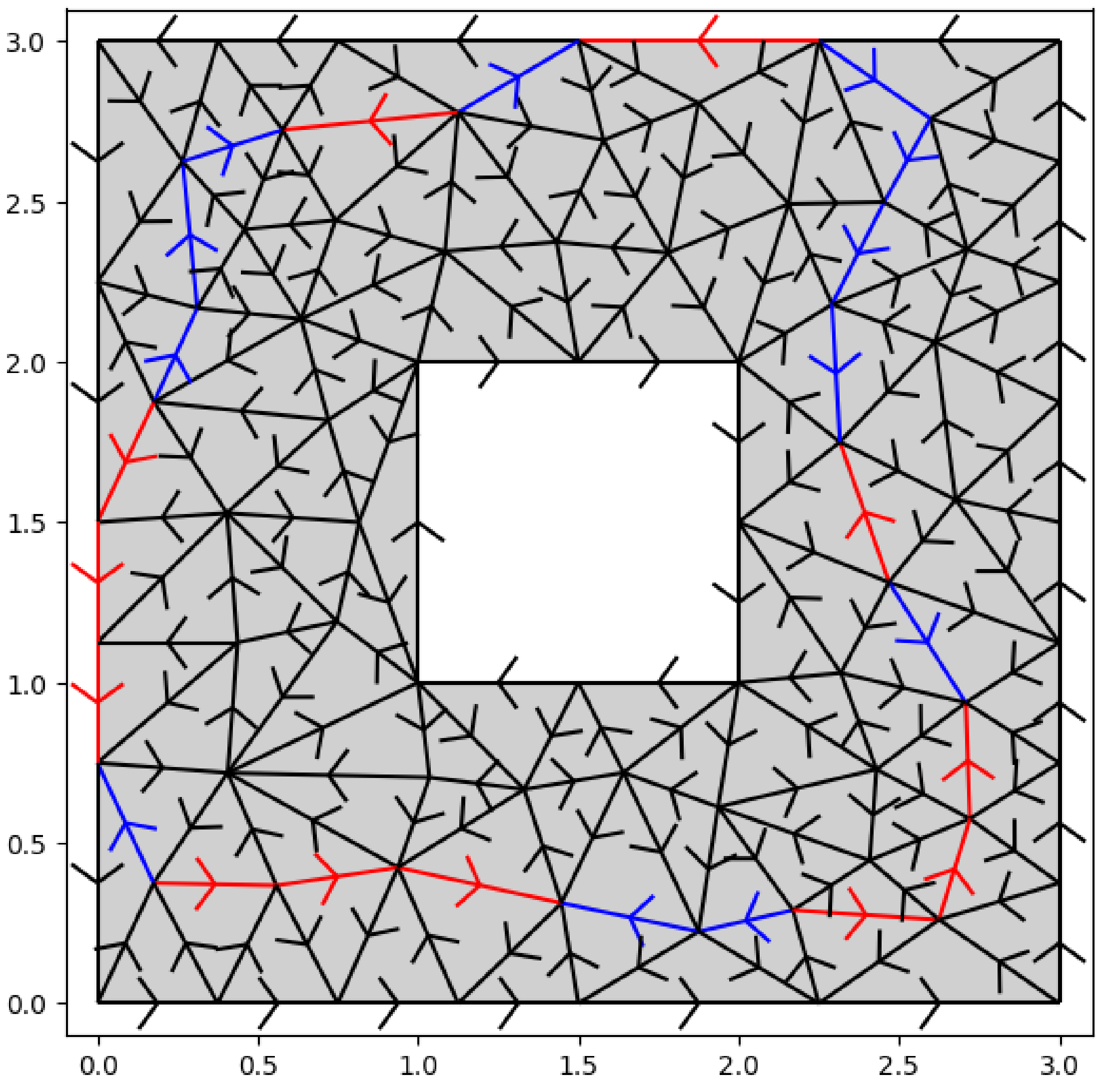
```
B2Dense=B2.todense()

print(nEdges-np.linalg.matrix_rank(B1Dense))
print(np.linalg.matrix_rank(B2Dense))
```

```
120
119
```

[37]:
```
# there is one equivalence class of cycles that is NOT a boundary!
# intuitively clear: these are the paths that go around the hole
```

[38]:
```
# construct such a path and examine it in more detail
dist,pred=getShortestPaths(pointData,edgeData)
```

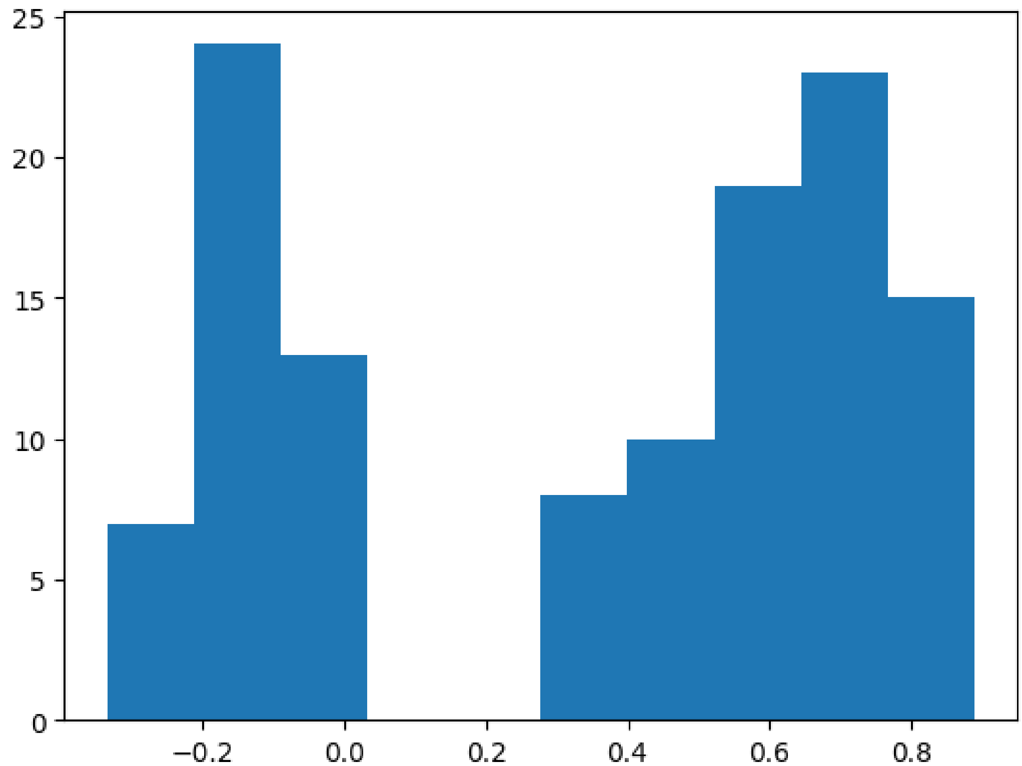
[39]:
```
# first build a cycle (which we conjecture to be a boundary)
# fix some vertices, connect shortest paths between them
iList=[18,34,52,63,18]
path=np.zeros(nEdges)
for j in range(len(iList)-1):
    pathList=extractPath(pred,iList[j],iList[j+1])
    path+=translatePath(pathList,edgeData)


fig=plt.figure(figsize=(6,6),facecolor="w")
ax=fig.add_subplot(aspect=1.)

polyCollection=matplotlib.collections.
 ↪LineCollection(pointData[triangleData],lw=0,fc="#d0d0d0",zorder=-2)
ax.add_collection(polyCollection)
lineColors=signToColor(path)
drawEdges(ax,pointData,edgeData,colors=lineColors,drawOrientations=True,arrowScale=0.
 ↪1,zorder=1)

#for i,x in enumerate(pointData):
#    ax.text(x[0],x[1],i,c="r")

# plot range
buffer=0.1
lim1,lim2,lim3,lim4=np.min(pointData[:,0]),np.max(pointData[:,0]),np.
 ↪min(pointData[:,1]),np.max(pointData[:,1])
plt.xlim([lim1-buffer,lim2+buffer])
plt.ylim([lim3-buffer,lim4+buffer])


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



[40]:
```python
# as above: try to find region which has given path as boundary
# by trying to solve solve: B2.dot(region)=path
# since we do not know for sure if solution exists, use leastsquares ansatz
res=scipy.sparse.linalg.lsqr(B2,path)
region=res[0]
# test solution quality: this time we have a non-zero residual!
print("norm of discrepancy between B2(region) and path:",res[3])
print("if solution was found, this should be 1: ",res[1])
```

```
norm of discrepancy between B2(region) and path: 1.4968407965017811
if solution was found, this should be 1:  2
```

[41]:
```
# if a good solution was found, then region should be binary, with only 0 and 1␣
↪entries
plt.hist(region)
plt.show()
```



[42]:
```
# now add the boundary of the hole as additional column to B2:

# first: extract bounday of hole
iList=[4,5,7,6,4]
holeBoundary=np.zeros(nEdges)
for j in range(len(iList)-1):
    pathList=extractPath(pred,iList[j],iList[j+1])
    holeBoundary+=translatePath(pathList,edgeData)

# now add to B2:
B2Ext=scipy.sparse.hstack((B2,scipy.sparse.csr_matrix(holeBoundary.
↪reshape((-1,1)))))
```

[43]:
```
# do the rank test again:

# recall:
# -image of B2 lives in kernel of B1
```

```
# -compare dimensions:
#   dim(ker B1)=nEdges-rank(B1)
#   dim(img B2)=rank(B2)

# since we use small matrices: determine rank on dense matrices
B1Dense=B1.todense()
B2Dense=B2Ext.todense()

print(nEdges-np.linalg.matrix_rank(B1Dense))
print(np.linalg.matrix_rank(B2Dense))
```

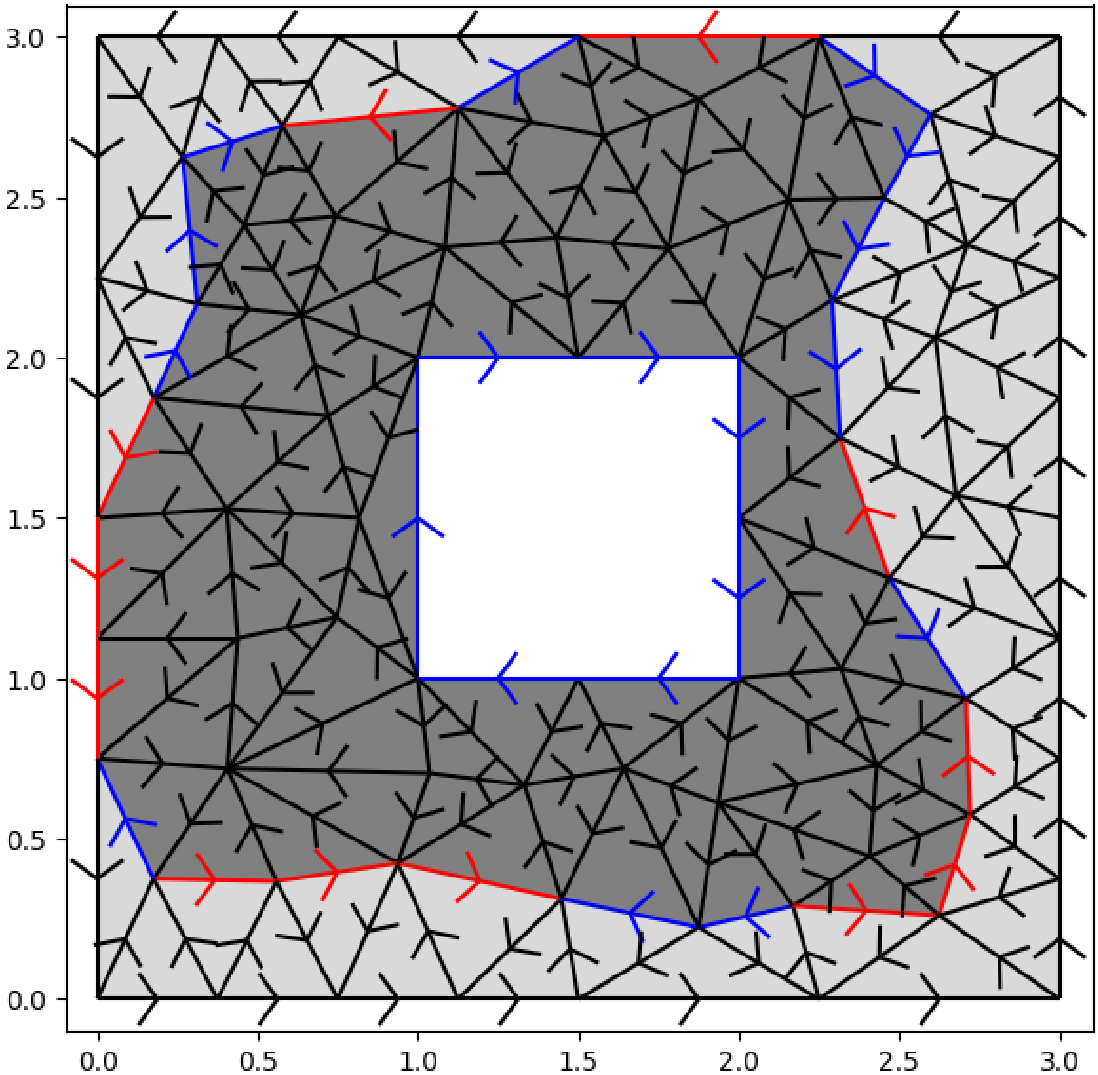
```
120
120
```

[44]:
```
# do least squares solving again:
res=scipy.sparse.linalg.lsqr(B2Ext,path)
region=res[0]
# test solution quality:
print("norm of discrepancy between B2(region) and path:",res[3])
print("if solution was found, this should be 1: ",res[1])
```

```
norm of discrepancy between B2(region) and path: 6.776446353336523e-05
if solution was found, this should be 1:  1
```

[45]:
```
iList=[18,34,52,63,18]
path=np.zeros(nEdges)
for j in range(len(iList)-1):
    pathList=extractPath(pred,iList[j],iList[j+1])
    path+=translatePath(pathList,edgeData)

pathExt=path.copy()
pathExt+=holeBoundary*region[-1]

fig=plt.figure(figsize=(6,6),facecolor="w")
ax=fig.add_subplot(aspect=1.)


regionColors=np.zeros((nTriangles,3))
regionColors[...]=np.array([0.85,0.85,0.85])

#
regionColors[(region[:-1]>0.1),:]=np.array([0.5,0.5,0.5])
polyCollection=matplotlib.collections.
 ↪LineCollection(pointData[triangleData],lw=0,fc=regionColors,zorder=-2)
ax.add_collection(polyCollection)
lineColors=signToColor(pathExt)
```

```
drawEdges(ax,pointData,edgeData,colors=lineColors,drawOrientations=True,arrowScale=0.
↪1,zorder=1)

#for i,x in enumerate(pointData):
#    ax.text(x[0],x[1],i,c="r")

# plot range
buffer=0.1
lim1,lim2,lim3,lim4=np.min(pointData[:,0]),np.max(pointData[:,0]),np.
↪min(pointData[:,1]),np.max(pointData[:,1])
plt.xlim([lim1-buffer,lim2+buffer])
plt.ylim([lim3-buffer,lim4+buffer])


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



1.10 Now repeat this on the 2-torus

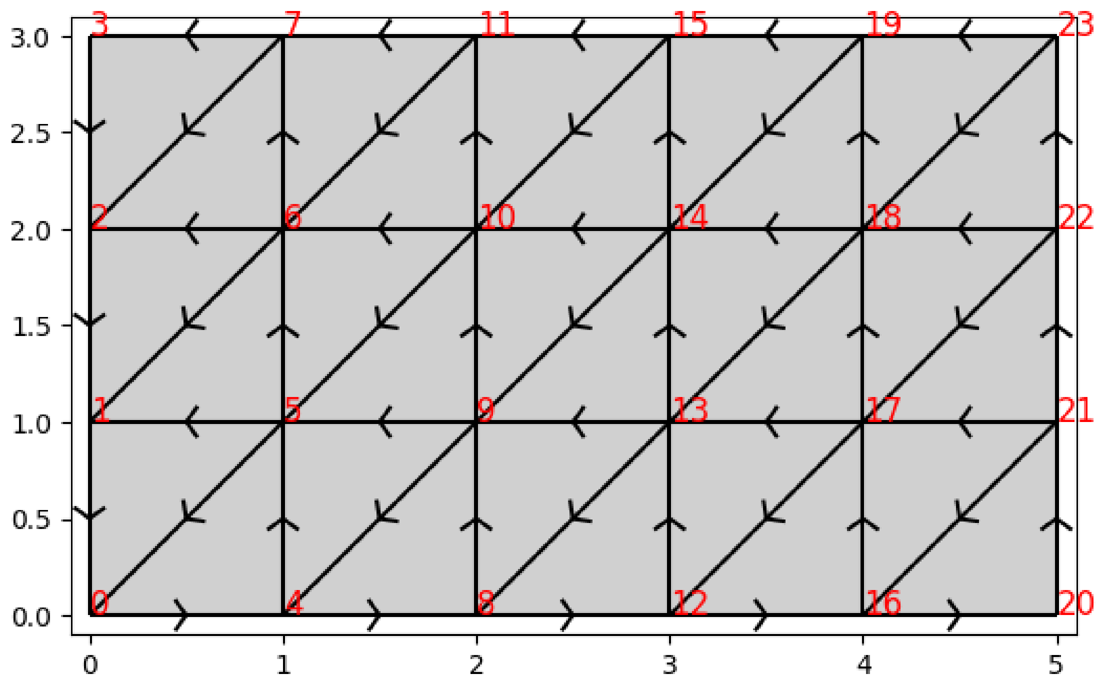
[46]:
```
def getPoslistNCube(shape,dtype=np.double):
    """Create list of positions in an n-dimensional cuboid of size shape."""
    ndim=len(shape)

    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()
```

[47]:
```
# first build regular triangulation of rectangle
nRows=4
nCols=6

pointData=getPoslistNCube((nCols,nRows))

nTriangles=(nRows-1)*(nCols-1)*2
triangleData=np.zeros((nTriangles,3),dtype=np.int32)
k=0
for i in range(nCols-1):
    for j in range(nRows-1):
        ind0=i*nRows+j
        ind1=(i+1)*nRows+j
        ind2=(i+1)*nRows+(j+1)
        ind3=i*nRows+(j+1)
        triangleData[k]=np.array([ind0,ind1,ind2])
        k+=1
        triangleData[k]=np.array([ind0,ind2,ind3])
        k+=1

nPoints=pointData.shape[0]

# extract edge data
edgeData,etAdjacencyData,etAdjacencyDataOrientation=FEM.getEdges(triangleData)
nEdges=edgeData.shape[0]
```

[48]:
```
%matplotlib inline
fig=plt.figure(figsize=(6,6),facecolor="w")
ax=fig.add_subplot(aspect=1.)

polyCollection=matplotlib.collections.
 ↪LineCollection(pointData[triangleData],lw=0,fc="#d0d0d0",zorder=-2)
ax.add_collection(polyCollection)
drawEdges(ax,pointData,edgeData,colors="k",drawOrientations=True,arrowScale=0.1)

for i,x in enumerate(pointData):
    ax.text(x[0],x[1],i,c="r",size=12)

# plot range
buffer=0.1
lim1,lim2,lim3,lim4=np.min(pointData[:,0]),np.max(pointData[:,0]),np.
 ↪min(pointData[:,1]),np.max(pointData[:,1])
plt.xlim([lim1-buffer,lim2+buffer])
```

```
plt.ylim([lim3-buffer,lim4+buffer])


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



[49]:
```
# Euler characteristic is that of planar mesh (not counting the "rest of␣
↪space"-face)
nTriangles-nEdges+nPoints
```

[49]: 1

[50]:
```
# now build mesh for torus
nRows=10
nCols=20

nTriangles=nRows*nCols*2
triangleData=np.zeros((nTriangles,3),dtype=np.int32)
k=0
for i in range(nCols):
    for j in range(nRows):
        ind0=i*nRows+j
        ind1=((i+1)%nCols)*nRows+j
        ind2=((i+1)%nCols)*nRows+((j+1)%nRows)
```

```
        ind3=i*nRows+((j+1)%nRows)
        triangleData[k]=np.array([ind0,ind1,ind2])
        k+=1
        triangleData[k]=np.array([ind0,ind2,ind3])
        k+=1

nPoints=nRows*nCols

# extract edge data
edgeData,etAdjacencyData,etAdjacencyDataOrientation=FEM.getEdges(triangleData)
nEdges=edgeData.shape[0]
```

[51]:
```
# Euler characteristic:
nTriangles-nEdges+nPoints
```

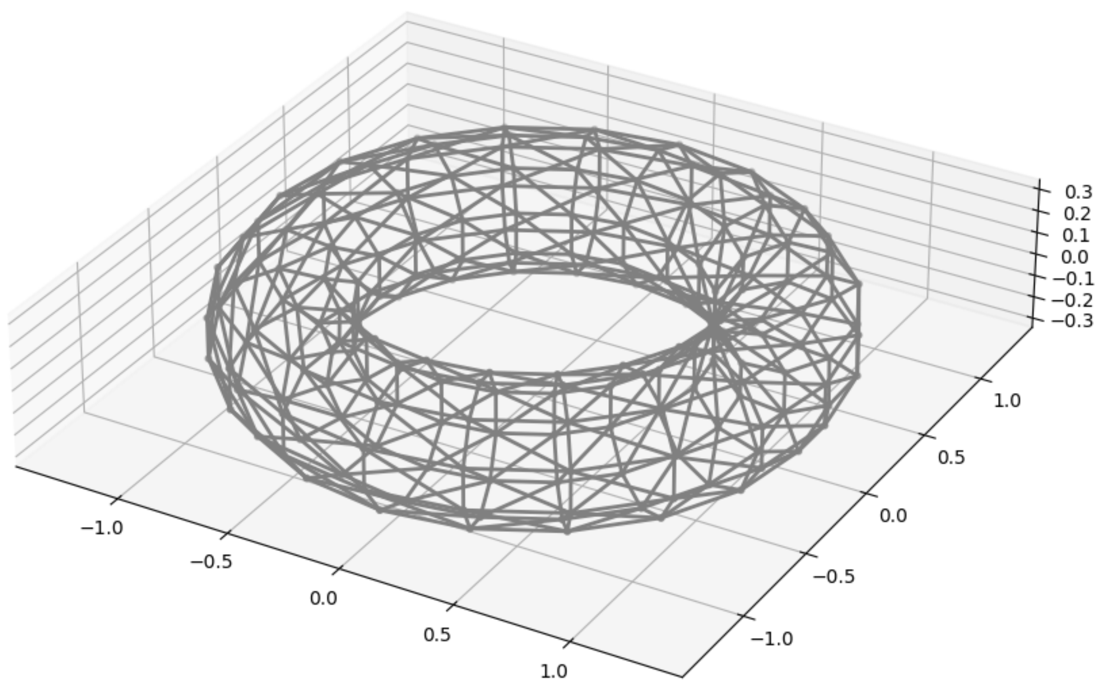
[51]: 0

[52]:
```
# at this point the mesh is only abstract, we have no coordinates
# now build the well-known "donut embedding"
pointData=np.zeros((nPoints,3),dtype=np.double)

# big and small radii:
R=1.
r=0.3
k=0
for i in range(nCols):
    for j in range(nRows):
        # "large angle"
        phi=i/nCols*2*np.pi
        theta=j/nRows*2*np.pi

        n1=np.array([np.cos(phi),np.sin(phi),0.])
        n2=np.array([0.,0.,1.])
        pointData[k,:]=R*n1+r*np.cos(theta)*n2+r*np.sin(theta)*n1
        k+=1
```

[53]:
```
import mpl_toolkits
```

[56]:
```
%matplotlib inline
fig = plt.figure(figsize=(8,8))

ax = fig.add_subplot(projection='3d')

ax.scatter(pointData[:,0],pointData[:,1],pointData[:,2],s=10,c="#808080")
lineCollection=mpl_toolkits.mplot3d.art3d.
 ↪Line3DCollection(pointData[edgeData],color="#808080",lw=2)
ax.add_collection(lineCollection)
```

```
ax.set_box_aspect((R+r,R+r,r))

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



[57]:
```
B1=getB1(nPoints,edgeData)
B2=getB2(nEdges,etAdjacencyData,etAdjacencyDataOrientation)
```

[58]:
```
# do the rank test again:

# recall:
# -image of B2 lives in kernel of B1
```

```python
# -compare dimensions:
#   dim(ker B1)=nEdges-rank(B1)
#   dim(img B2)=rank(B2)

# since we use small matrices: determine rank on dense matrices
B1Dense=B1.todense()
B2Dense=B2.todense()

print(nEdges-np.linalg.matrix_rank(B1Dense))
print(np.linalg.matrix_rank(B2Dense))
```

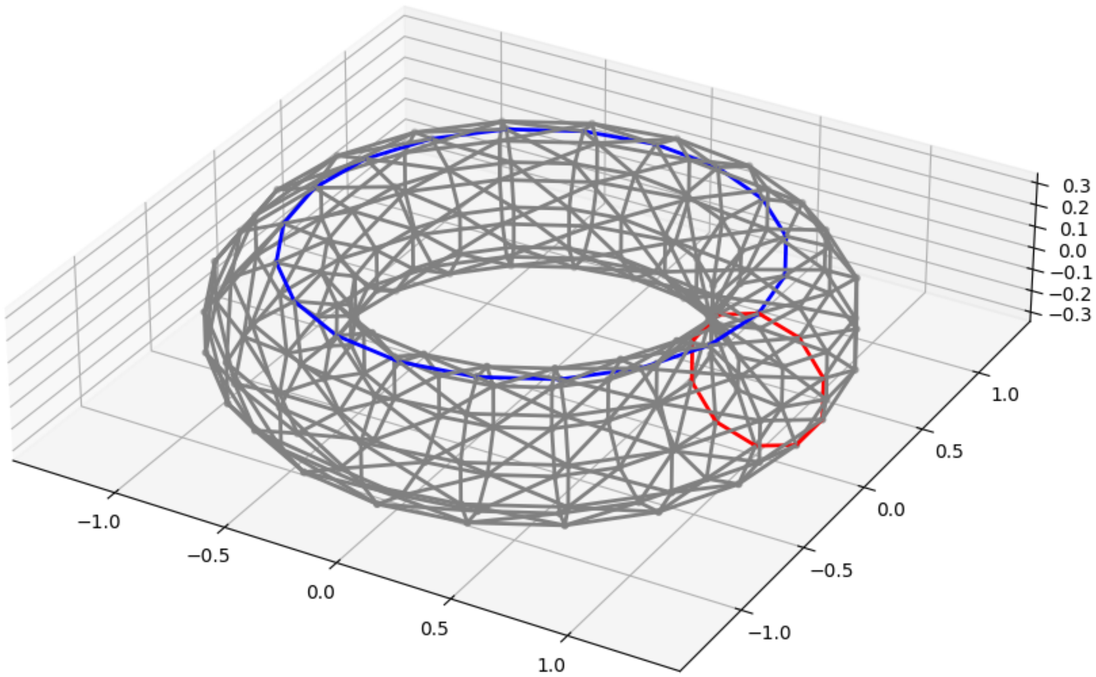
```
401
399
```

[59]:

```python
path1Pre=np.arange(nRows+1)
path1Pre[-1]=0
path1=translatePath(path1Pre,edgeData)

path2Pre=np.arange(nCols+1)*nRows
path2Pre[-1]=0
path2=translatePath(path2Pre,edgeData)
```

[62]:

```python
%matplotlib inline
fig = plt.figure(figsize=(8,8))

ax = fig.add_subplot(projection='3d')

ax.scatter(pointData[:,0],pointData[:,1],pointData[:,2],s=10,c="#808080")

lineColors=np.zeros((nEdges,3))
lineColors[...]=np.array([0.5,0.5,0.5])
lineColors[(np.abs(path1[:])>0.1),:]=np.array([1.,0.,0.])
lineColors[(np.abs(path2[:])>0.1),:]=np.array([0.,0.,1.])

lineCollection=mpl_toolkits.mplot3d.art3d.
 ↪Line3DCollection(pointData[edgeData],color=lineColors,lw=2)
ax.add_collection(lineCollection)


ax.set_box_aspect((R+r,R+r,r))

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



[63]:
```
# add these as extra columns to B2:
B2Ext=scipy.sparse.hstack((B2,\
        scipy.sparse.csr_matrix(path1.reshape((-1,1))),\
        scipy.sparse.csr_matrix(path2.reshape((-1,1)))\
        ))
```

[64]:
```
# do the rank test again:

# recall:
# -image of B2 lives in kernel of B1
# -compare dimensions:
#   dim(ker B1)=nEdges-rank(B1)
#   dim(img B2)=rank(B2)
```

```
# since we use small matrices: determine rank on dense matrices
B1Dense=B1.todense()
B2Dense=B2Ext.todense()

print(nEdges-np.linalg.matrix_rank(B1Dense))
print(np.linalg.matrix_rank(B2Dense))
```

```
401
401
```

[65]:

```
B1.dot(B2Ext)
```

[65]:

```
<200x402 sparse matrix of type '<class 'numpy.float64'>'
        with 0 stored elements in Compressed Sparse Row format>
```

[ ]:

```

```