Robust Successive Binarizations (RSB) for Change Detection in Hyperspectral Images

In [1]:

```
#Necessary libraries to be imported:
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import time
from os.path import dirname, join as pjoin
from scipy.io import loadmat
from sklearn.metrics import confusion_matrix
import cv2
from sklearn.naive_bayes import GaussianNB
from scipy import stats
```

LOAD the DATASETS

The followin dataset have been downloaded from here:

- Hermiston: https://gitlab.citius.usc.es/hiperespectral/ChangeDetectionDataset/-/tree/master/Hermiston (https://gitlab.citius.usc.es/hiperespectral/ChangeDetectionDataset/-/tree/master/Hermiston)
- USA: https://rslab.ut.ac.ir/documents/81960329/82034892/Hyperspectral_Change_Datasets.zip (https://rslab.ut.ac.ir/documents/81960329/82034892/Hyperspectral_Change_Datasets.zip)
- River: http://crabwq.github.io (http://crabwq.github.io)
- Bay Area: https://gitlab.citius.usc.es/hiperespectral/ChangeDetectionDataset/-/tree/master/bayArea (https://gitlab.citius.usc.es/hiperespectral/ChangeDetectionDataset/-/tree/master/bayArea)

Plase, replace the following `filepath`, `filepathU`, `filepathR`, `filepathB` string with your own.

In [2]:

```
filepath = 'C:/Users/antonella/Downloads/ChangeDetectionDataset-master-Hermiston/Change
her1 = pjoin(filepath,'hermiston2004.mat')
her2 = pjoin(filepath,'hermiston2007.mat')
gt = pjoin(filepath,'rdChangesHermiston_5classes.mat')
```

In [3]:

```
filepathU = 'C:/Users/antonella/Downloads/Hyperspectral_Change_Datasets'
usa1 = pjoin(filepathU,'USA1.mat')
usa2 = pjoin(filepathU,'USA2.mat')
gtU = pjoin(filepathU,'USA_gt.mat')
```

In [4]:

```
filepathR = 'C:/Users/antonella/Downloads/GETNET/zuixin'
river1 = pjoin(filepathR,'river_before.mat')
river2 = pjoin(filepathR,'river_after.mat')
gtR = pjoin(filepathR,'groundtruth.mat')
```

In [5]:

```
filepathB = 'C:/Users/antonella/Documents/all_Desktop/CalcNUM_Bari/SALIENCY/Graziano_Sa
bay1 = pjoin(filepathB,'Bay_Area_2013.mat')
bay2 = pjoin(filepathB,'Bay_Area_2015.mat')
gtB = pjoin(filepathB,'bayArea_gtChangesolf.mat')
```

In [6]:

```
her1 = loadmat(her1) # Dictionary
her1 = her1['HypeRvieW']
her1 = her1.astype(float)
her2 = loadmat(her2) # Dictionary
her2 = her2['HypeRvieW']
her2 = her2.astype(float)

usa1 = loadmat(usa1)
usa1 = usa1['USA1']
usa1 = usa1.astype(float)
usa2 = loadmat(usa2) # Dictionary
usa2 = usa2['USA2']
usa2 = usa2.astype(float)

river1 = loadmat(river1)
river1 = river1['river_before']
river1 = river1.astype(float)
river2 = loadmat(river2) # Dictionary
river2 = river2['river_after']
river2 = river2.astype(float)

bay1 = loadmat(bay1)
bay1 = bay1['HypeRvieW']#
bay1 = bay1.astype(float)
bay2 = loadmat(bay2) # Dictionary
bay2 = bay2['HypeRvieW']
bay2 = bay2.astype(float)
```

In [7]:

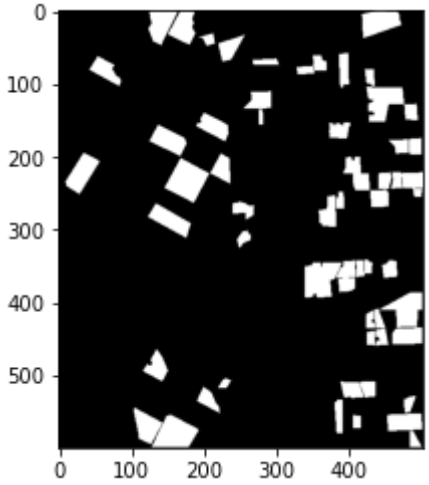
```
gt = loadmat(gt)
gt = gt['gt5clasesHermiston']
gt = gt.astype(float)

gtU = loadmat(gtU)
gtU = gtU['USA_gt']
gtU = gtU.astype(float)

gtR = loadmat(gtR)
gtR = gtR['lakelabel_v1']
gtR = gtR.astype(float)

gtB = loadmat(gtB)
gtB = gtB['HypeRvieW']
gtB = gtB.astype(float)

```

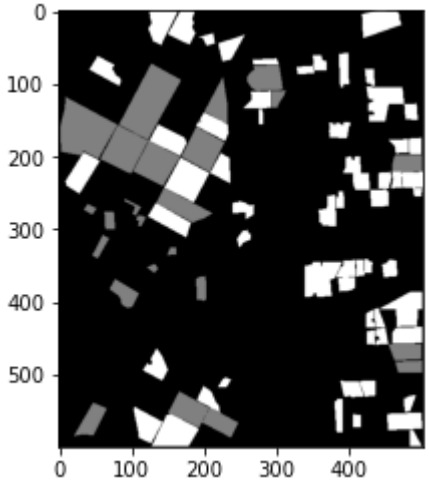
In [8]:

```
[mB,nB]= gtB.shape
label_2 = np.argwhere(gtB==2)#global indices of label 2 pixels

NgtB = np.copy(gtB)
Printing = np.copy(gtB)

NgtB[label_2[:,0], label_2[:,1]] = 0
plt.figure()
plt.imshow(NgtB.reshape(mB,nB),cmap = plt.cm.gray)

Printing[label_2[:,0], label_2[:,1]] = 0.5
plt.figure()
plt.imshow(Printing, cmap = plt.cm.gray)
```

Out[8]:

```
<matplotlib.image.AxesImage at 0x1b4a0142f88>
```





In [9]:

```
# Hermiston GT is given for multi-class, hence in order to produce only the binary map
# changed/not-changed pixels, we need to set to 1 all the changed pixels

n2 = np.where(gt.ravel()==2)
n3 = np.where(gt.ravel()==3)
n4 = np.where(gt.ravel()==4)
n5 = np.where(gt.ravel()==5)

new_gt = gt.ravel()
######## &&&& Only for  Hermiston
new_gt[n2] = 1
new_gt[n3] = 1
new_gt[n4] = 1
new_gt[n5] = 1
```

In [27]:

```
[m1,n1,k1] = her1.shape
print('Hermiston dataset size ', m1,n1,k1)

H1 = np.reshape(her1,[m1*n1,k1])
print('Hermiston vectorized ', H1.shape)

[m2,n2,k2] = her2.shape

H2 = np.reshape(her2,[m2*n2,k2])

[mu1,nu1,ku1] = usa1.shape
print('USA dataset size  ', mu1,nu1,ku1)

U1 = np.reshape(usa1,[mu1*nu1,ku1])
print('USA vectorized ', U1.shape)

[mu2,nu2,ku2] = usa2.shape

U2 = np.reshape(usa2,[mu2*nu2,ku2])

[mr1,nr1,kr1] = river1.shape
print('River dataset size  ', mr1,nr1,kr1)

R1 = np.reshape(river1,[mr1*nr1,kr1])
print('River vectorized ', R1.shape)

[mr2,nr2,kr2] = river2.shape

R2 = np.reshape(river2,[mr2*nr2,kr2])

[mb1,nb1,kb1] = bay1.shape
print('Bay dataset size  ', mb1,nb1,kb1)

B1 = np.reshape(bay1,[mb1*nb1,kb1])
[mb2,nb2,kb2] = bay2.shape

B2 = np.reshape(bay2,[mb2*nb2,kb2])
print('Bay vectorized ', B1.shape)

```

```
Hermiston dataset size  390 200 242
Hermiston vectorized  (78000, 242)
USA dataset size   307 241 154
USA vectorized  (73987, 154)
River dataset size   463 241 198
River vectorized  (111583, 198)
Bay dataset size   600 500 224
Bay vectorized  (300000, 224)
```

In [28]:

```
for i in range(mb1*nb1):
    MB1 = np.max(B1[i,:])
    mB1= np.min(B1[i,:])
    MB2 = np.max(B2[i,:])
    mB2= np.min(B2[i,:])
    # y -m1 = m(x-m2)
    #   y- m1 = (M1-m1)/(M2-m2)(x-m2)
    #    M1-m1 = m(M2-m2)

    B2[i,:] = (B2[i,:] -mB2)*(MB1-mB1)/(MB2-mB2)+mB1


for i in range(ku1):
    MU1 = np.max(U1[:,i])
    mU1= np.min(U1[:,i])
    MU2 = np.max(U2[:,i])
    mU2= np.min(U2[:,i])
    U2[:,i] = (U2[:,i] -mU2)*(MU1-mU1)/(MU2-mU2)+mU1

print('\n DONE \n ')

```

```
 DONE
```

In [29]:

```
Old1 = np.copy(B1)
Old2 = np.copy(B2)
```

In [177]:

```
B1 = np.copy(Old1)
B2 = np.copy(Old2)
```

In [178]:

```
B1 = abs(B1)**0.2
B2 = abs(B2)**0.2

```

Error Function Definitions

The fucntions: SAM-MEAN, newSSCC (i.e., SMSADM) and SAM_g_ZID have been introduced here:

-Falini, A., Tamborrino, C., Castellano, G., Mazzia, F., Mininni, R. M., Appice, A., & Malerba, D. (2020, July). Novel reconstruction errors for saliency detection in hyperspectral images. In International Conference on Machine Learning, Optimization, and Data Science (pp. 113-124). Springer, Cham.

In [12]:

```
def SAM(x,y):
    '''
    INPUT:
    x and y are two mono-dimensional arrais.
    OUTPUT:
    scalar value
    '''

    norm2_x = np.linalg.norm(x)
    norm2_y = np.linalg.norm(y)
    return np.arccos(np.dot(x,y)/(norm2_x*norm2_y))


def SAM_MEAN(A, B, windowSize=2):
    '''
    INPUT:
    A and B are 3-mode tensors with size rows x col x feature

    OUTPUT:
    a matrix of size rows x col
    '''



    row, col, feature= A.shape
    C= np.zeros([row, col])
    windSizeMatrix=np.zeros((row, col))

    for i in range(-windowSize, +windowSize+1):
        for j in range(-windowSize, +windowSize+1):
            rowS=max(0,i)
            rowE=min(row,row+i)
            colS=max(0,j)
            colE=min(col,col+j)
            windSizeMatrix[row-rowE:row-rowS, col-colE:col-colS]+=1
            den1= np.sqrt(np.sum(np.multiply(A[rowS:rowE, colS:colE], A[rowS:rowE, colS
            den1[den1 < 1e-5]=1e-5

            den2= np.sqrt(np.sum(np.multiply(B[rowS:rowE, colS:colE], B[rowS:rowE, colS
            den2[den2 < 1e-5]=1e-5
            num=np.sum(np.multiply(A[rowS:rowE, colS:colE],B[rowS:rowE, colS:colE]),axi
            ndiv = np.divide(num, np.multiply(den1,den2))
            ndiv[ndiv > 1] = 1.
            ndiv[ndiv <-1] = -1.
            count=np.arccos(ndiv)

            C[row-rowE:row-rowS, col-colE:col-colS]+=count


    C=np.divide(C, windSizeMatrix).reshape(row,col)

    return C
```

In [13]:

```
def newSSCC(A, B, windowSize=2):
    '''
    INPUT:
    A and B are 3-mode tensors with size rows x col x feature

    OUTPUT:
    a matrix of size rows x col
    '''
    row, col, feature= A.shape

    nPixel=row*col
    sumA = A.sum(axis=2)
    sumB = B.sum(axis=2)

    BwindowMean=np.zeros(sumB.shape)
    numT=np.zeros(sumA.shape)
    den2T=np.zeros(sumA.shape)
    denT=np.zeros(sumA.shape)

    AwindowMean=np.zeros(sumA.shape)
    windSizeMatrix=np.zeros(sumA.shape)
    den1T=np.zeros(sumA.shape)

    for i in range(-windowSize, +windowSize+1):
        for j in range(-windowSize, +windowSize+1):
                rowS=max(0,i)
                rowE=min(row,row+i)
                colS=max(0,j)
                colE=min(col,col+j)
                AwindowMean[row-rowE:row-rowS, col-colE:col-colS]+=sumA[rowS:rowE, colS

                BwindowMean[row-rowE:row-rowS, col-colE:col-colS]+=sumB[rowS:rowE, colS
                windSizeMatrix[row-rowE:row-rowS, col-colE:col-colS]+=1
        #windSizeMatrix=np.multiply(windSizeMatrix,windSizeMatrix)
        AwindowMean=AwindowMean/(feature)
        #print('\n max AwindowMean ', np.max(AwindowMean))
        BwindowMean=BwindowMean/(feature)
        #print('\n max BwindowMean ', np.max(BwindowMean))
        #print('\n max windSizeMatrix ', np.max(windSizeMatrix))
        BwindowMean=np.divide(BwindowMean, windSizeMatrix)
        AwindowMean=np.divide(AwindowMean, windSizeMatrix)
        #print('\n First division ')

        for i in range(-windowSize, +windowSize+1):
            for j in range(-windowSize, +windowSize+1):
                rowS=max(0,i)
                rowE=min(row,row+i)
                colS=max(0,j)
                colE=min(col,col+j)
                Asubtract=np.zeros(A.shape)
                Bsubtract=np.zeros(A.shape)


                Asubtract[rowS:rowE, colS:colE]=A[rowS:rowE, colS:colE]-AwindowMean[row
                Bsubtract[rowS:rowE, colS:colE]=B[rowS:rowE, colS:colE]-BwindowMean[row

                den1T[row-rowE:row-rowS, col-colE:col-colS]+=np.einsum('ijn,ijn->ij', A

                numT[row-rowE:row-rowS, col-colE:col-colS]+=np.einsum('ijn,ijn->ij', As
```

```
60                 den2T[row-rowE:row-rowS, col-colE:col-colS]+=np.einsum('ijn,ijn->ij', E
61
62         denT=np.multiply(np.sqrt(den1T),np.sqrt(den2T))
63         denT[denT< 1e-5]=1e-5
64         #print('\n max den1T ', np.max(den1T))
65         #print('\n max den2T ', np.max(den2T))
66         C=1-np.divide(numT,denT).reshape(row, col)
67         #print('\n second division ')
68         return C
69
```

In [14]:

```
def nor01(matrix):
    mi=(matrix.min())
    ma=(matrix).max()
    return (matrix-mi)/(ma-mi)

def Scale(Matrix):
    minMat = np.min(Matrix)
    MaxMat = np.max(Matrix)
    return 1/(MaxMat-minMat)*(Matrix-minMat)

def SAM_ZID(A,B):
    '''
    INPUT:
    A and B are 3-mode tensors with size rows x col x feature

    OUTPUT:
    a matrix of size rows x col
    '''
    row, column, feature= A.shape
    nPixel=row*column
    sam = np.zeros([1,nPixel])
    Ac = A.reshape(nPixel,feature)
    Bc = B.reshape(nPixel,feature)
    for i in range(nPixel):
        sam[0,i] = SAM(Ac[i,:], Bc[i,:])

    diffe = abs(Ac-Bc)
    temp = np.zeros([nPixel, feature])
    for k in range(nPixel):
        temp[k,:] = ((diffe[k,:]-np.mean(diffe[k,:]))/np.std(diffe[k,:]))**2
    zid = np.sum(temp, axis=1)
    sin_angle = nor01( np.sin( sam ))
    zidj = nor01(zid)
    sam_zid= np.prod([[sin_angle[0,:],zidj]], axis=1)

    return sam_zid

def ZID_g_mod(A,B):
    '''
    INPUT:
    A and B are 3-mode tensors with size rows x col x feature

    OUTPUT:
    a matrix of size rows x col
    '''

    row, column, feature= A.shape
    nPixel=row*column
    A= (A.reshape(nPixel,feature))
    B = B.reshape(nPixel,feature)
    mu_g0=np.zeros([nPixel,1] )
    sigma_g0 = np.zeros([nPixel,1] )
    muB=np.zeros([nPixel,1] )
    sigmaB = np.zeros([nPixel,1] )
    muA=np.zeros([nPixel,1] )
    sigmaA = np.zeros([nPixel,1] )
    C0 =np.copy(abs(A-B)) # modified

    eta = 1e-10
```

```
    mu_g0[:,0] = ( (np.sum(np.log(( C0+eta)),axis=1)/A.shape[1]) )

    sigma_g0[:,0] = ( np.sum((( C0+eta-mu_g0)**2), axis=1)/(A.shape[1]) )
    sigma_g0[sigma_g0 < 1e-10]=1e-5

    zid_g = np.sum( (C0-(np.exp(mu_g0)))**2/( ((sigma_g0)) ),axis=1)
    return zid_g

def SAM_g_ZID(A,B):
    '''
    INPUT:
    A and B are 3-mode tensors with size rows x col x feature

    OUTPUT:
    a matrix of size rows x col
    '''
    row, column, feature= A.shape
    nPixel=row*column
    sam = np.zeros([1,nPixel])
    Ac = A.reshape(nPixel,feature)
    Bc = B.reshape(nPixel,feature)
    for i in range(nPixel):
        sam[0,i] = SAM(Ac[i,:], Bc[i,:])

    zid =(ZID_g_mod(A,B))
    sin_angle = nor01( np.tan( sam ))
    zidj = nor01(zid)
    sam_zid= np.prod([[sin_angle[0,:],zidj]],axis=1)

    return sam_zid


def OA_K(cnf):
    index = np.nonzero(cnf)
    TN, FP, FN, TP=cnf[index]
    OA = (TP+TN)/(TP+TN+FP+FN)
    p = ((TP + FP)*(TP+FN)+(TN+FN)*(TN+FP))/(TP + TN + FP + FN)**2
    Kappa = (OA-p)/(1-p)
    return OA,Kappa

def Binarize1(M):
    '''
    INPUT:
    M is a matrix, i.e., a bidimensional array

    OUTPUT:
    a flattened matrix
    '''
    Temp = np.copy(M)
    Temp2 = np.copy(Temp)
    Temp3 = np.copy(Temp)
    Temp4 = np.copy(Temp)
    Temp5 = np.copy(Temp)
    Temp6 = np.copy(Temp)


    Temp2[Temp>=0.2]=1
    Temp2[Temp<0.2]=0

    Temp3[Temp>=0.3]=1
    Temp3[Temp<0.3]=0
```

```
121
122     Temp4[Temp>=0.4]=1
123     Temp4[Temp<0.4]=0
124
125     Temp5[Temp>=0.5]=1
126     Temp5[Temp<0.5]=0
127
128     Temp6[Temp>=0.6]=1
129     Temp6[Temp<0.6]=0
130
131     MediaM = (Temp2+Temp3+Temp4+ Temp5+Temp6)/2.0
132
133     return MediaM
```

Euclidean distance

In [179]:

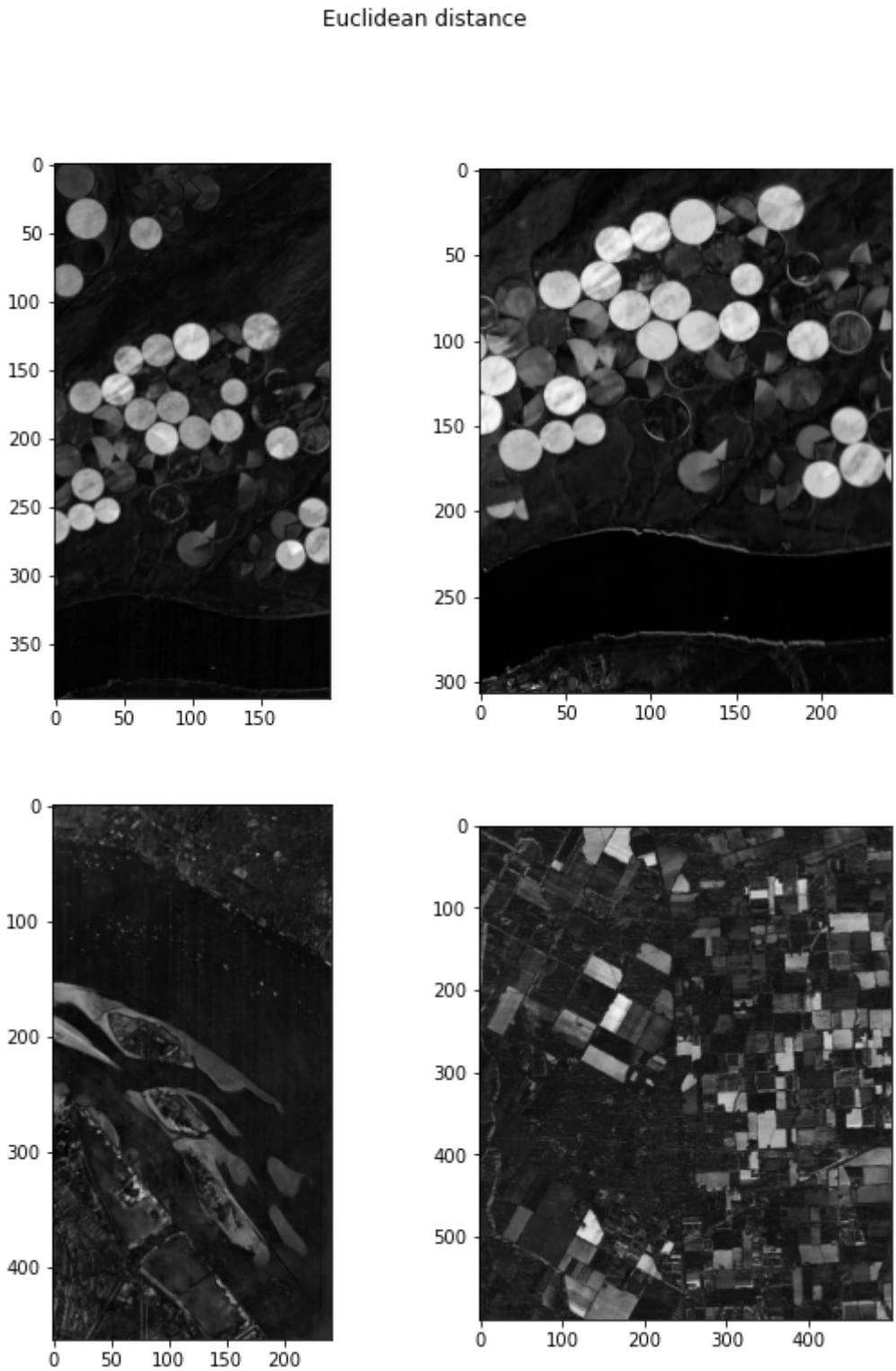
```
 1 diff_matH = H1-H2
 2 EuH = np.zeros([1,m1*n1])
 3 for i in range(m1*n1):
 4     EuH[0,i] = np.linalg.norm(diff_matH[i,:])
 5
 6
 7 diff_matU = U1-U2
 8 EuU = np.zeros([1,mu1*nu1])
 9 for i in range(mu1*nu1):
10     EuU[0,i] = np.linalg.norm(diff_matU[i,:])
11
12
13 diff_matR = R1-R2
14 EuR = np.zeros([1,mr1*nr1])
15 for i in range(mr1*nr1):
16     EuR[0,i] = np.linalg.norm(diff_matR[i,:])
17
18 diff_matB = B1-B2
19 EuB = np.zeros([1,mb1*nb1])
20 for i in range(mb1*nb1):
21     EuB[0,i] = np.linalg.norm(diff_matB[i,:])
```

In [180]:

```
fig, axarr=plt.subplots(nrows=2, ncols=2, figsize=(9, 12))
fig.suptitle('Euclidean distance')
axarr[0,0].imshow(EuH.reshape(m1,n1),cmap = plt.cm.gray)
axarr[0,1].imshow(EuU.reshape(mu1,nu1),cmap = plt.cm.gray)
axarr[1,0].imshow(EuR.reshape(mr1,nr1),cmap = plt.cm.gray)
axarr[1,1].imshow(EuB.reshape(mb1,nb1),cmap = plt.cm.gray)
```

Out[180]:

```
<matplotlib.image.AxesImage at 0x1b4eaaa2b48>
```



We apply binarization:

- We scale the matrix values between 0 and 1
- We apply several thresholds

In [181]:

```
EuH = Scale(EuH)

EuU = Scale(EuU)

EuR = Scale(EuR)

EuB = Scale(EuB)
```

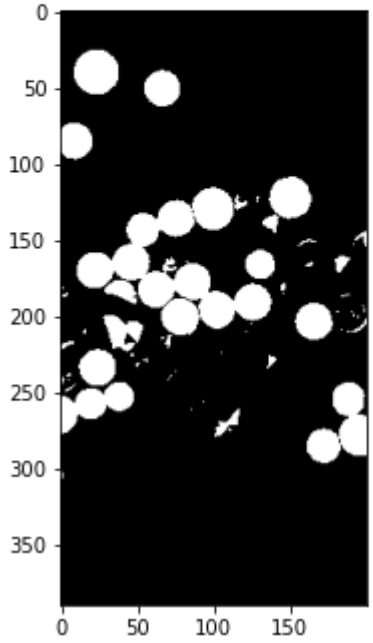
In [182]:

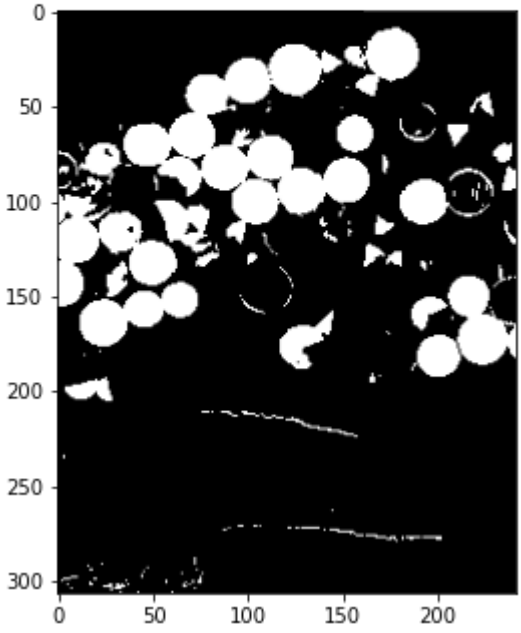
```
MediaEuH = Binarize1(EuH)

MediaEuH[MediaEuH>=1.0]=1.0
MediaEuH[MediaEuH<1.0]=0.0
MediaEuH = MediaEuH.astype(int)

MediaEuU = Binarize1(EuU)

MediaEuU[MediaEuU>=1.0]=1.0
MediaEuU[MediaEuU<1.0]=0.0
MediaEuU = MediaEuU.astype(int)

MediaEuR = Binarize1(EuR)
MediaEuR[MediaEuR>=1.0]=1.0
MediaEuR[MediaEuR<1.0]=0.0
MediaEuR = MediaEuR.astype(int)

MediaEuB = Binarize1(EuB)
MediaEuB[MediaEuB>=1.0]=1.0
MediaEuB[MediaEuB<1.0]=0.0
MediaEuB = MediaEuB.astype(int)



fig, axarr=plt.subplots(nrows=2, ncols=2, figsize=(9, 12))
fig.suptitle('Median Euclidean distance')
axarr[0,0].imshow(MediaEuH.reshape(m1,n1),cmap = plt.cm.gray)
axarr[0,1].imshow(MediaEuU.reshape(mu1,nu1),cmap = plt.cm.gray)
axarr[1,0].imshow(MediaEuR.reshape(mr1,nr1),cmap = plt.cm.gray)
axarr[1,1].imshow(MediaEuB.reshape(mb1,nb1),cmap = plt.cm.gray)
#axarr[1,1].imshow(MediaEuBa.reshape(mba1,nba1),cmap = plt.cm.gray)
```

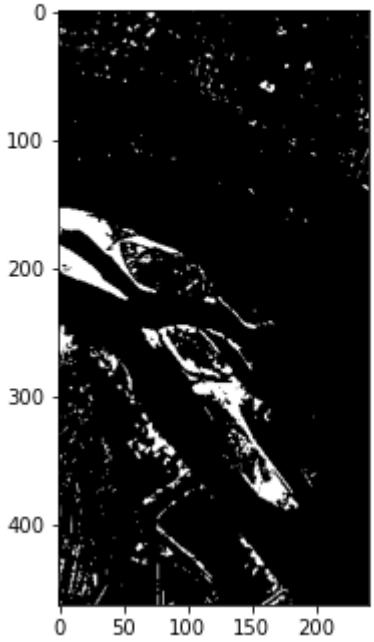
Out[182]:

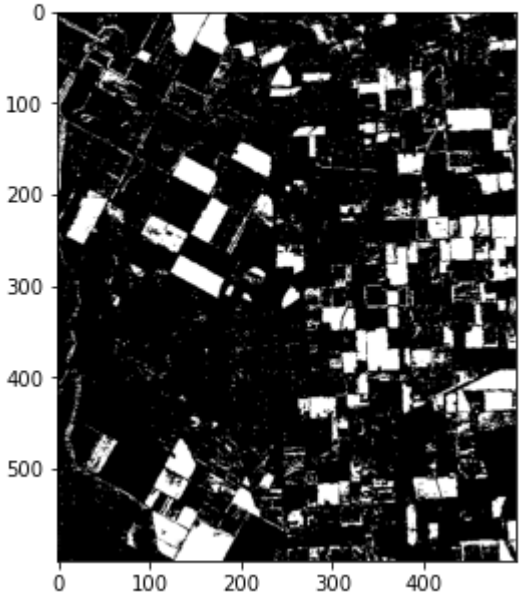
```
<matplotlib.image.AxesImage at 0x1b4eb1f3608>
```

Median Euclidean distance









SAMZID

In [183]:

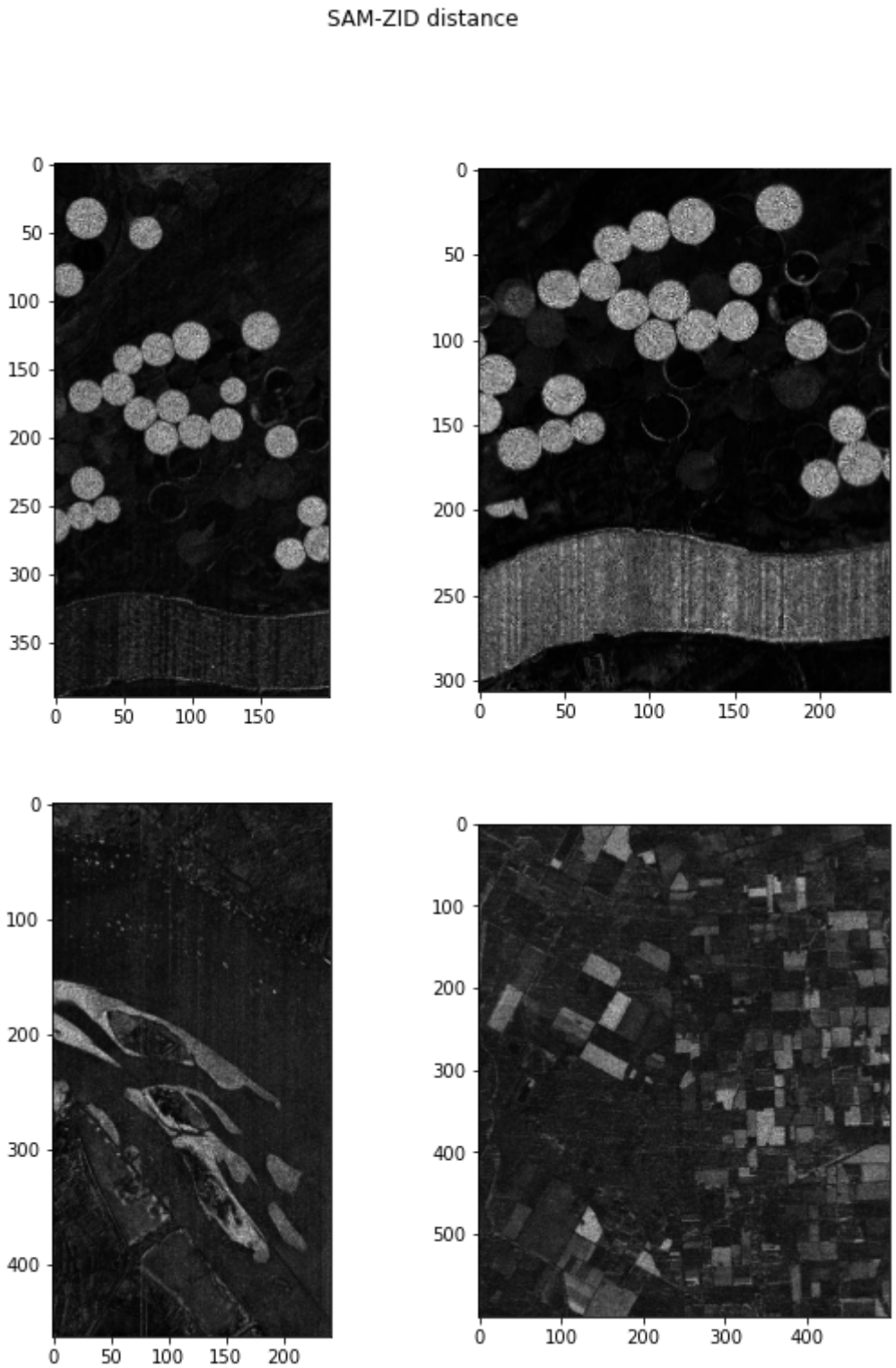
```
Hssz = SAM_ZID(H1.reshape(m1,n1,k1),H2.reshape(m2,n2,k2))

Ussz = SAM_ZID(U1.reshape(mu1,nu1,ku1),U2.reshape(mu2,nu2,ku2))

Rssz = SAM_ZID(R1.reshape(mr1,nr1,kr1),R2.reshape(mr2,nr2,kr2))

Bssz = SAM_ZID(B1.reshape(mb1,nb1,kb1),B2.reshape(mb2,nb2,kb2))
```

In [184]:

```
fig, axarr=plt.subplots(nrows=2, ncols=2, figsize=(9, 12))
fig.suptitle('SAM-ZID distance')
axarr[0,0].imshow(Hssz.reshape(m1,n1),cmap = plt.cm.gray)
axarr[0,1].imshow(Ussz.reshape(mu1,nu1), cmap = plt.cm.gray)
axarr[1,0].imshow(Rssz.reshape(mr1,nr1), cmap = plt.cm.gray)
axarr[1,1].imshow(Bssz.reshape(mb1,nb1), cmap = plt.cm.gray)
```

Out[184]:

<matplotlib.image.AxesImage at 0x1b4ecc89488>



In [185]:

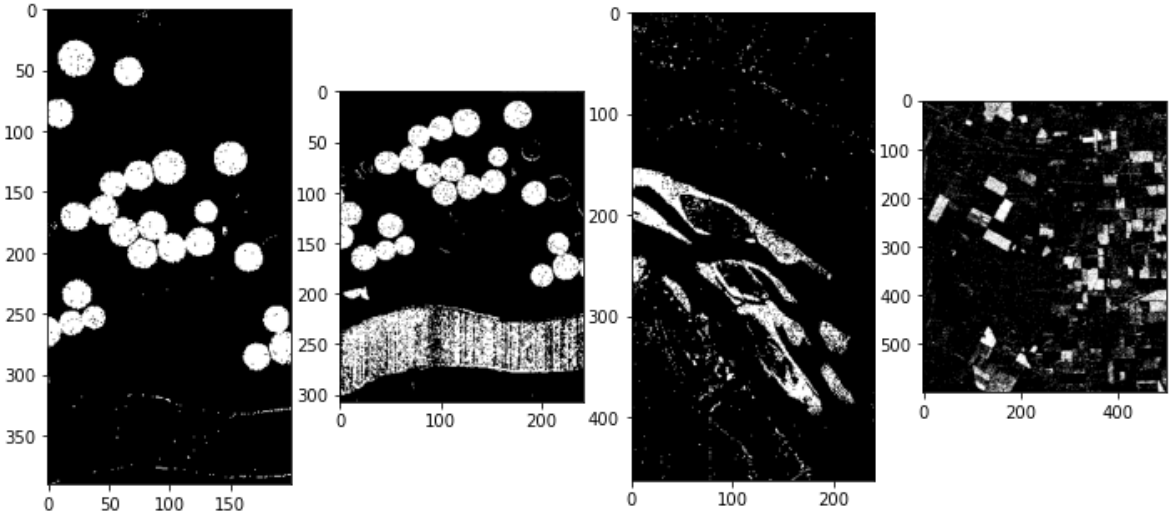
```
Hssz = Scale(Hssz)

Ussz = Scale(Ussz)

Rssz = Scale(Rssz)

Bssz = Scale(Bssz)
```

In [186]:

```
MediaHssz = Binarize1(Hssz)
MediaHssz[MediaHssz>=1.0]=1.0
MediaHssz[MediaHssz<1.0]=0.0
MediaHssz = MediaHssz.astype(int)


MediaUssz =  Binarize1(Ussz)
MediaUssz[MediaUssz>=1.0]=1.0
MediaUssz[MediaUssz<1.0]=0.0
MediaUssz = MediaUssz.astype(int)


MediaRssz =  Binarize1(Rssz)
MediaRssz[MediaRssz>=1.0]=1.0
MediaRssz[MediaRssz<1.0]=0.0
MediaRssz= MediaRssz.astype(int)

MediaBssz =  Binarize1(Bssz)
MediaBssz[MediaBssz>=1.0]=1.0
MediaBssz[MediaBssz<1.0]=0.0
MediaBssz= MediaBssz.astype(int)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4,figsize=(12,9))
fig.suptitle('Mean SAM-ZID distance')
ax1.imshow(MediaHssz.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(MediaUssz.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(MediaRssz.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(MediaBssz.reshape(mb1,nb1), cmap = plt.cm.gray)
```

Out[186]:

```
<matplotlib.image.AxesImage at 0x1b4f0222608>
```



SAM-MEAN

In [187]:

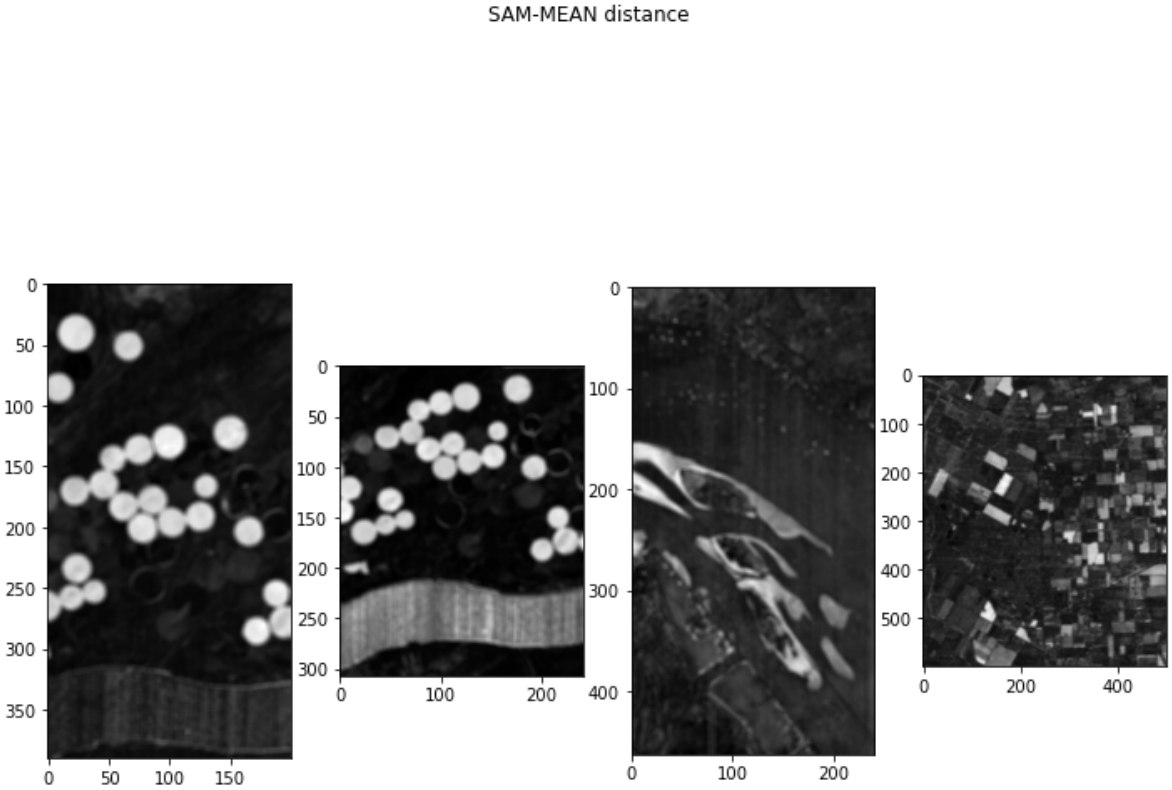
```
HsM = SAM_MEAN(H1.reshape(m1,n1,k1), H2.reshape(m2,n2,k2))

UsM = SAM_MEAN(U1.reshape(mu1,nu1,ku1), U2.reshape(mu2,nu2,ku2))

RsM = SAM_MEAN(R1.reshape(mr1,nr1,kr1), R2.reshape(mr2,nr2,kr2))

BsM = SAM_MEAN(B1.reshape(mb1,nb1,kb1), B2.reshape(mb2,nb2,kb2))
```

In [188]:

```
fig, (ax1, ax2, ax3,ax4) = plt.subplots(1, 4,figsize=(12,9))
fig.suptitle('SAM-MEAN distance')
ax1.imshow(HsM,cmap = plt.cm.gray)
ax2.imshow(UsM, cmap = plt.cm.gray)
ax3.imshow(RsM, cmap = plt.cm.gray)
ax4.imshow(BsM, cmap = plt.cm.gray)
```

Out[188]:

```
<matplotlib.image.AxesImage at 0x1b4ed3a77c8>
```



In [189]:

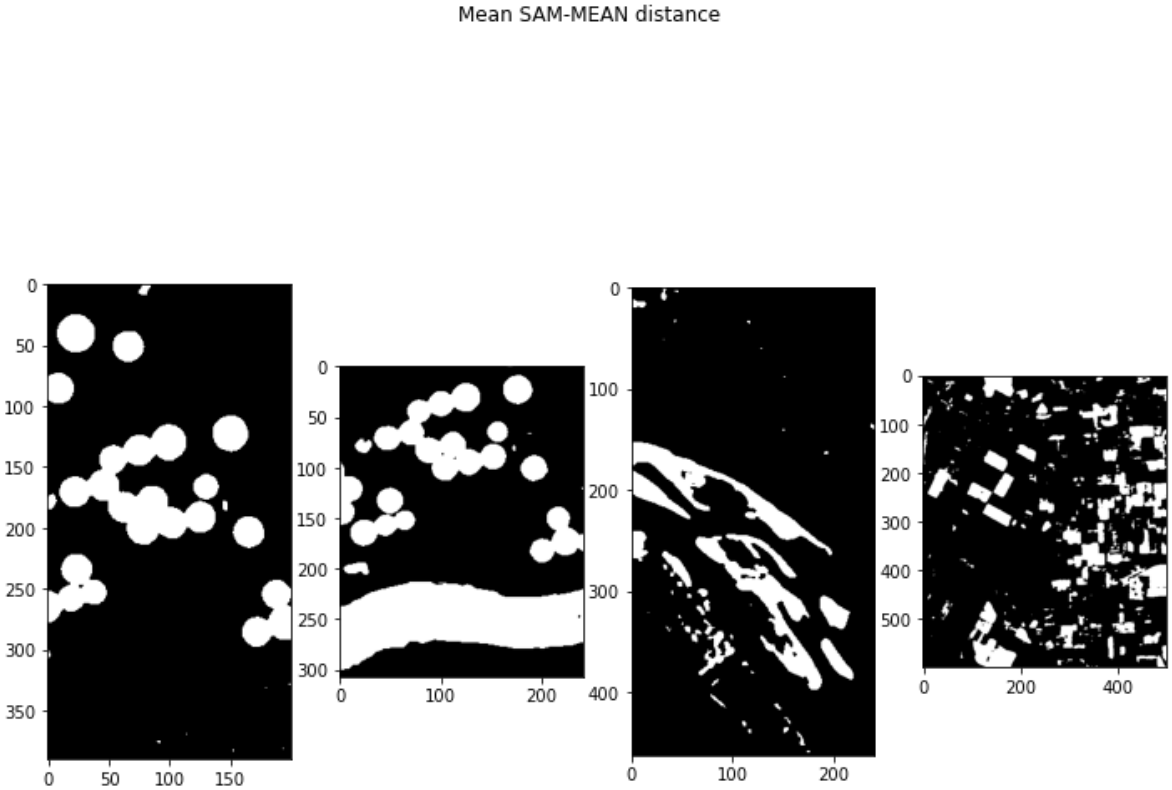
```
HsM = Scale(HsM)

UsM = Scale(UsM)

RsM = Scale(RsM)

BsM = Scale(BsM)
```

In [190]:

```
MediaHsM =  Binarize1(HsM)
MediaHsM[MediaHsM>=1.0]=1.0
MediaHsM[MediaHsM<1.0]=0.0
MediaHsM = MediaHsM.astype(int)


MediaUsM =  Binarize1(UsM)
MediaUsM[MediaUsM>=1.0]=1.0
MediaUsM[MediaUsM<1.0]=0.0
MediaUsM = MediaUsM.astype(int)


MediaRsM =  Binarize1(RsM)
MediaRsM[MediaRsM>=1.0]=1.0
MediaRsM[MediaRsM<1.0]=0.0
MediaRsM = MediaRsM.astype(int)


MediaBsM =  Binarize1(BsM)
MediaBsM[MediaBsM>=1.0]=1.0
MediaBsM[MediaBsM<1.0]=0.0
MediaBsM = MediaBsM.astype(int)

fig, (ax1, ax2, ax3,ax4) = plt.subplots(1, 4,figsize=(12,9))
fig.suptitle('Mean SAM-MEAN distance')
ax1.imshow(MediaHsM,cmap = plt.cm.gray)
ax2.imshow(MediaUsM, cmap = plt.cm.gray)
ax3.imshow(MediaRsM, cmap = plt.cm.gray)
ax4.imshow(MediaBsM, cmap = plt.cm.gray)
```

Out[190]:

```
<matplotlib.image.AxesImage at 0x1b4ed514e48>
```



In [29]:

```
'''To uncomment only if chunks of codes are run not in the prescribed order, as  dimens

#[m1,n1,k1] = her1.shape

#H1 = np.reshape(her1,[m1*n1,k1])

#[m2,n2,k2] = her2.shape

#H2 = np.reshape(her2,[m2*n2,k2])
```

Out[29]:

```
'To uncomment only if chunks of codes are run not in the prescribed order, a
s  dimensions might be different'
```

SMSADM

In [191]:

```
Hsc = newSSCC(H1.reshape(m1,n1,k1), H2.reshape(m2,n2,k2))

Usc = newSSCC(U1.reshape(mu1,nu1,ku1), U2.reshape(mu2,nu2,ku2))

Rsc = newSSCC(R1.reshape(mr1,nr1,kr1), R2.reshape(mr2,nr2,kr2))

Bsc = newSSCC(Old1.reshape(mb1,nb1,kb1), Old2.reshape(mb2,nb2,kb2))
```

...

In [192]:

```
indH = np.isnan(Hsc)
indU = np.isnan(Usc)
indR = np.isnan(Rsc)
indB = np.isnan(Bsc)
```

In [193]:

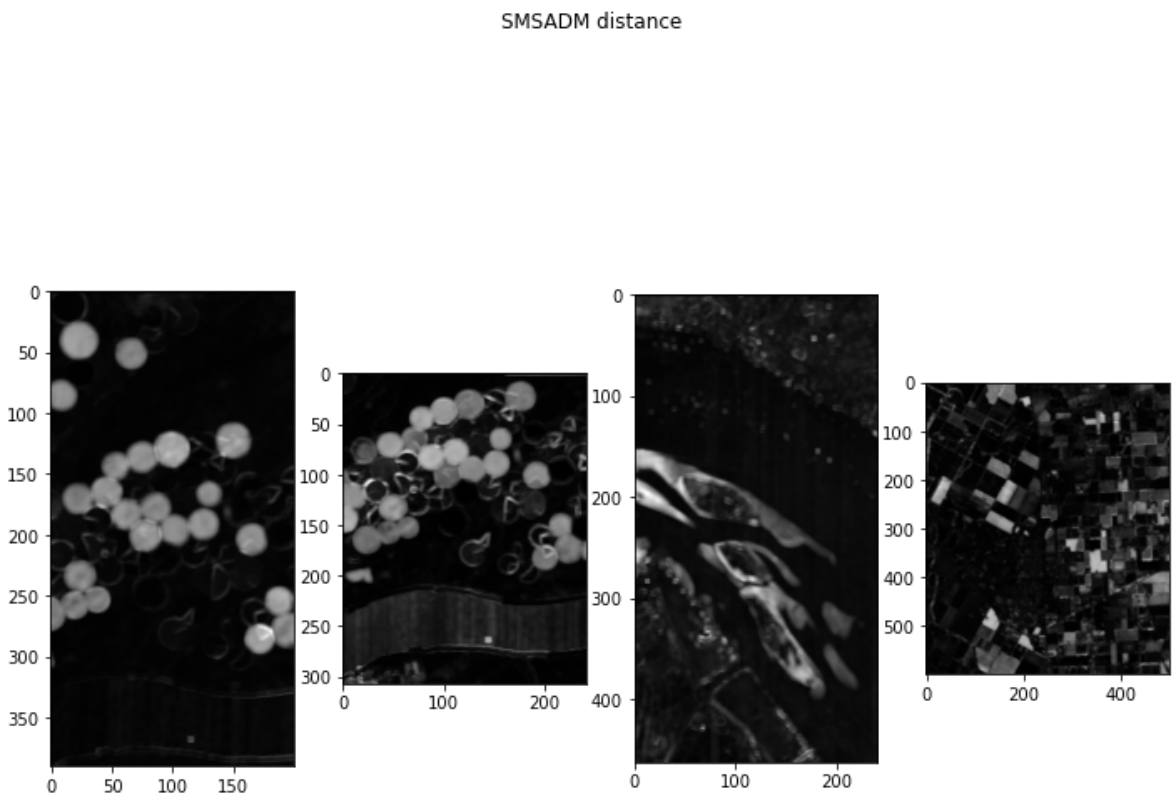
```
#  SMSADM might produces some sporious "0/0" due to numerical instabilities
Hsc[indH]=0.0

Usc[indU]=0.0

Rsc[indR]=0.0

Bsc[indB]=0.0
```

In [194]:

```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(12,9))
fig.suptitle('SMSADM distance')
ax1.imshow(Hsc.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(Usc.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(Rsc.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(Bsc.reshape(mb1,nb1), cmap = plt.cm.gray)
```

Out[194]:

```
<matplotlib.image.AxesImage at 0x1b4efb0c688>
```



In [195]:

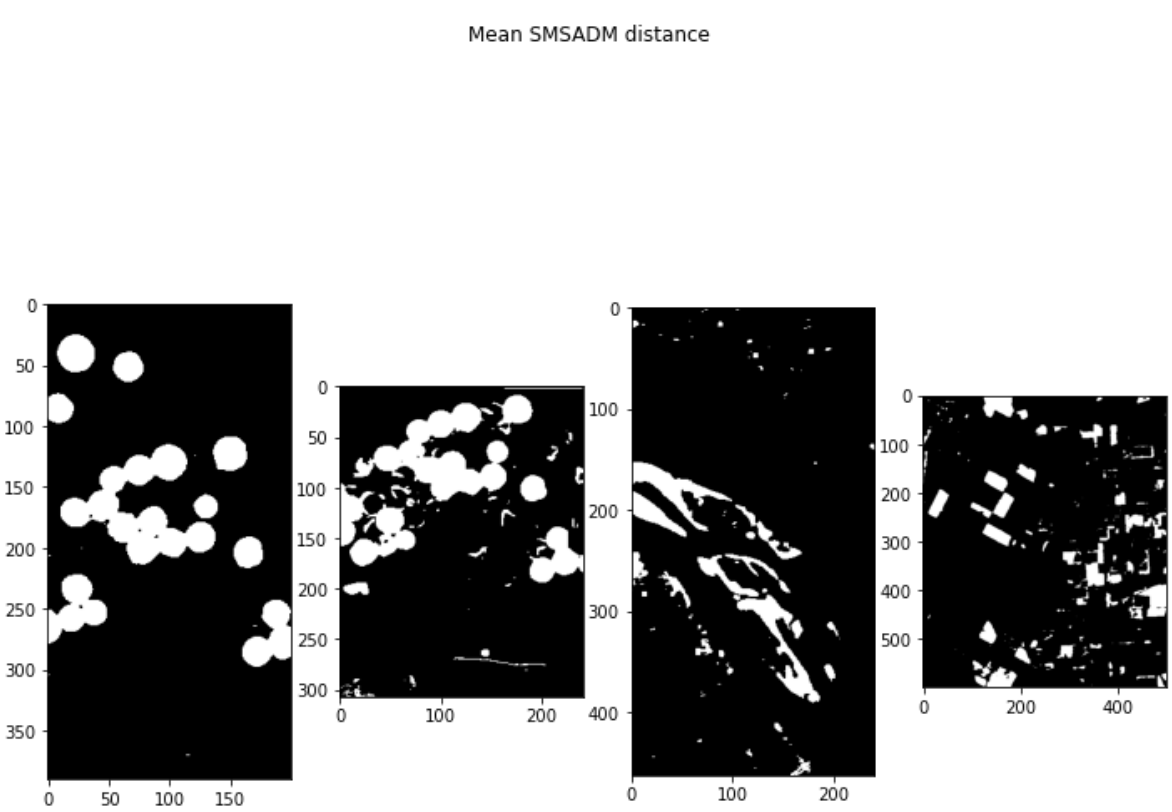
```
Hsc = Scale(Hsc)

Usc = Scale(Usc)

Rsc = Scale(Rsc)

Bsc = Scale(Bsc)
```

In [196]:

```
MediaHsc = Binarize1(Hsc)
MediaHsc[MediaHsc>=1.0]=1.0
MediaHsc[MediaHsc<1.0]=0.0
MediaHsc = MediaHsc.astype(int)

MediaUsc =  Binarize1(Usc)
MediaUsc[MediaUsc>=1.0]=1.0
MediaUsc[MediaUsc<1.0]=0.0
MediaUsc = MediaUsc.astype(int)


MediaRsc =  Binarize1(Rsc)
MediaRsc[MediaRsc>=1.0]=1.0
MediaRsc[MediaRsc<1.0]=0.0
MediaRsc = MediaRsc.astype(int)


MediaBsc =  Binarize1(Bsc)
MediaBsc[MediaBsc>=1.0]=1.0
MediaBsc[MediaBsc<1.0]=0.0
MediaBsc = MediaBsc.astype(int)

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(12,9))
fig.suptitle('Mean SMSADM distance')
ax1.imshow(MediaHsc.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(MediaUsc.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(MediaRsc.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(MediaBsc.reshape(mb1,nb1), cmap = plt.cm.gray)
```

Out[196]:

```
<matplotlib.image.AxesImage at 0x1b4efc791c8>
```



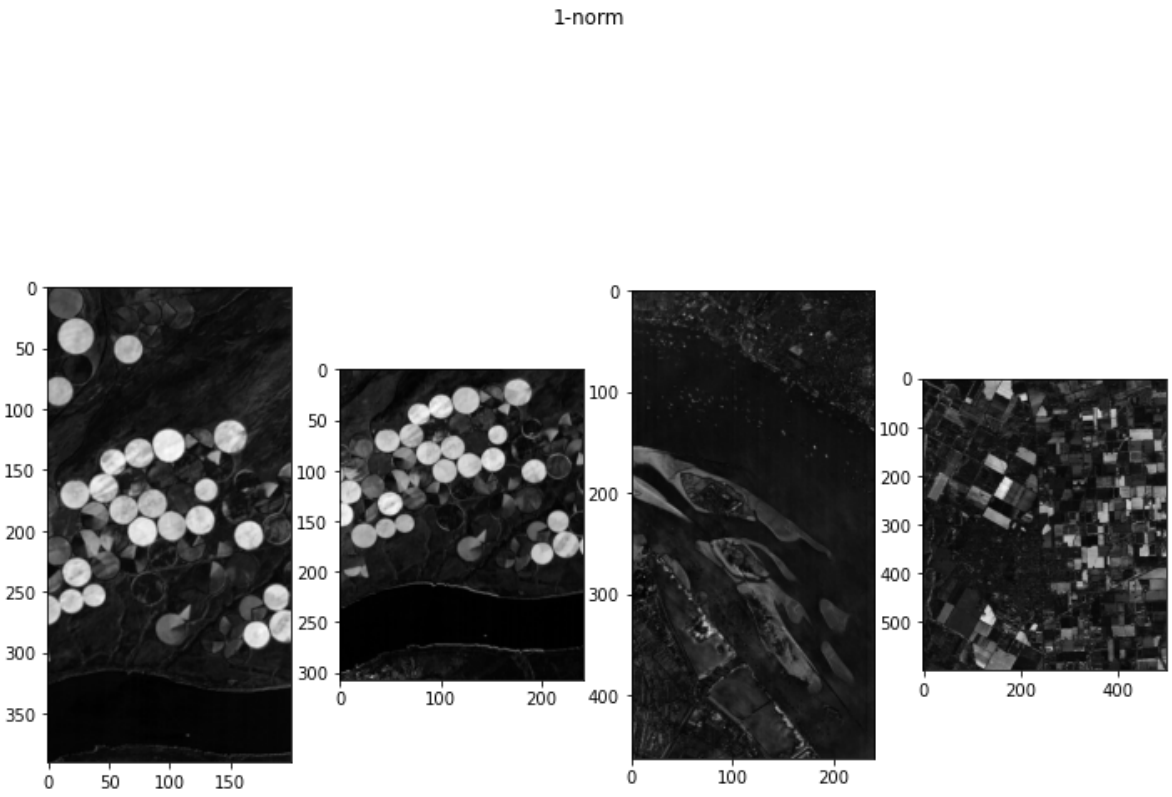
$\|\cdot\|_1$

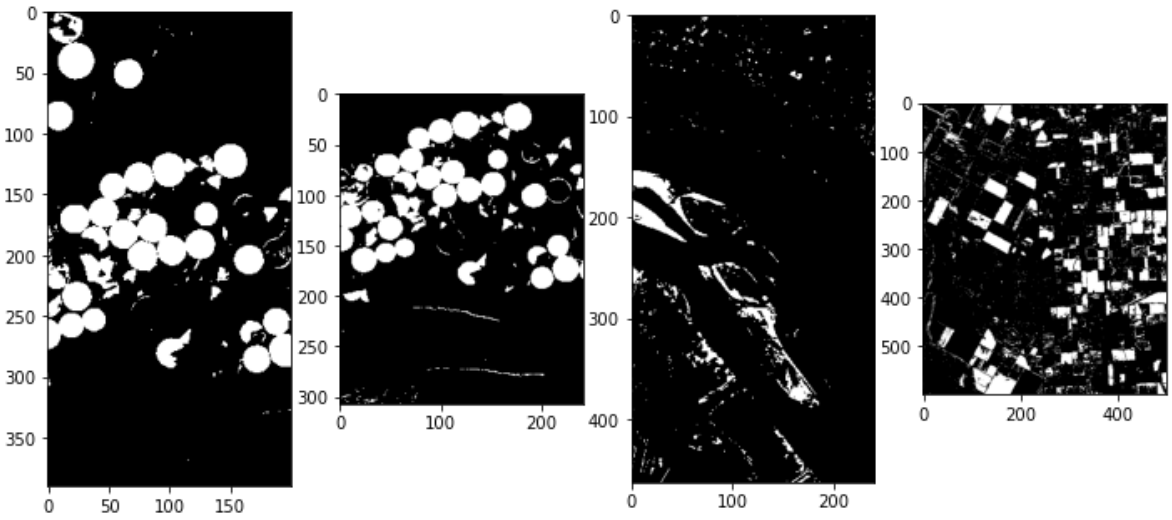
In [197]:

```
H1norm = np.zeros(m1*n1)
U1norm = np.zeros(mu1*nu1)
R1norm = np.zeros(mr1*nr1)
B1norm = np.zeros(mb1*nb1)


for i in range(m1*n1):
    H1norm[i]= np.sum(abs(H1[i,:]-H2[i,:]))
for i in range(mu1*nu1):
    U1norm[i]= np.sum(abs(U1[i,:]-U2[i,:]))
for i in range(mr1*nr1):
    R1norm[i]= np.sum(abs(R1[i,:]-R2[i,:]))
for i in range(mb1*nb1):
    B1norm[i]= np.sum(abs(B1[i,:]-B2[i,:]))

fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(12,9))
fig.suptitle('1-norm')
ax1.imshow(H1norm.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(U1norm.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(R1norm.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(B1norm.reshape(mb1,nb1), cmap = plt.cm.gray)

H1norm = Scale(H1norm)
U1norm = Scale(U1norm)
R1norm = Scale(R1norm)
B1norm = Scale(B1norm)


MediaH1norm = Binarize1(H1norm)
MediaH1norm[MediaH1norm>=1.0]=1.0
MediaH1norm[MediaH1norm<1.0]=0.0
MediaH1norm = MediaH1norm.astype(int)


MediaU1norm =  Binarize1(U1norm)
MediaU1norm[MediaU1norm>=1.0]=1.0
MediaU1norm[MediaU1norm<1.0]=0.0
MediaU1norm = MediaU1norm.astype(int)


MediaR1norm =  Binarize1(R1norm)
MediaR1norm[MediaR1norm>=1.0]=1.0
MediaR1norm[MediaR1norm<1.0]=0.0
MediaR1norm = MediaR1norm.astype(int)

MediaB1norm =  Binarize1(B1norm)
MediaB1norm[MediaB1norm>=1.0]=1.0
MediaB1norm[MediaB1norm<1.0]=0.0
MediaB1norm = MediaB1norm.astype(int)

#####

fig, (ax1, ax2, ax3,ax4) = plt.subplots(1, 4, figsize=(12,9))
fig.suptitle('Media 1-norm')
ax1.imshow(MediaH1norm.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(MediaU1norm.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(MediaR1norm.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(MediaB1norm.reshape(mb1,nb1), cmap = plt.cm.gray)
```

Out[197]:

```
<matplotlib.image.AxesImage at 0x1b4f091ea88>
```





In [198]:

```
def pearson_cc(v1,v2):
    '''
    INPUT:
    v1, v2 monodimensional arraies

    OUTPUT:
    scalar value between -1 and 1
    '''
    mv1 = np.mean(v1)
    mv2 = np.mean(v2)
    norm1 = np.linalg.norm(v1-mv1)
    norm2 = np.linalg.norm(v2-mv2)
    return np.dot((v1-mv1).T,(v2-mv2))/(norm1*norm2+1e-8)
```

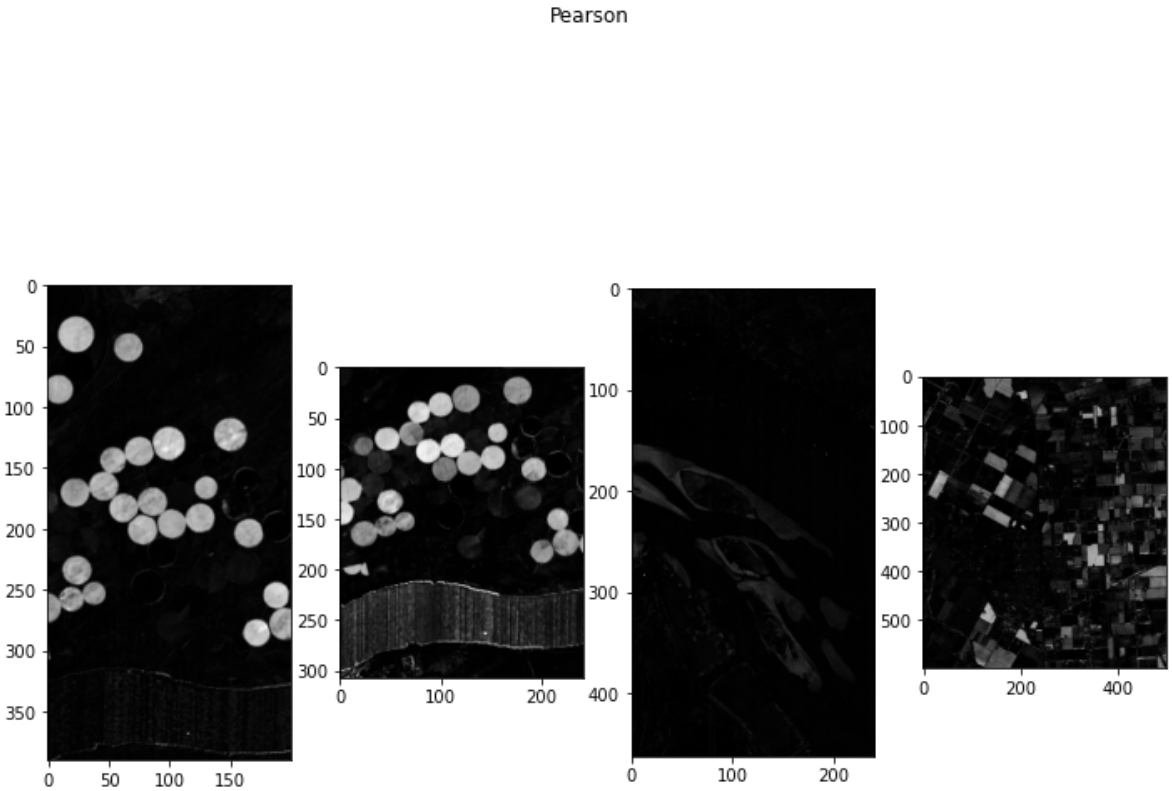
In [200]:

```
Hpr = np.zeros(m1*n1)
Upr = np.zeros(mu1*nu1)
Rpr = np.zeros(mr1*nr1)
Bpr = np.zeros(mb1*nb1)


for i in range(m1*n1):
    Hpr[i]= pearson_cc(H1[i,:], H2[i,:])#,sH = stats.pearsonr((H1[i,:]),(H2[i,:]))
for i in range(mu1*nu1):
    Upr[i]= pearson_cc(U1[i,:], U2[i,:])#,sU = stats.pearsonr((U1[i,:]),(U2[i,:]))
for i in range(mr1*nr1):
    Rpr[i]= pearson_cc(R1[i,:], R2[i,:])#,sR = stats.pearsonr((R1[i,:]),(R2[i,:]))

for i in range(mb1*nb1):
    Bpr[i]=pearson_cc(Old1[i,:], Old2[i,:])#,sB = stats.pearsonr((B1[i,:]),(B2[i,:]))#


Hpr = 1-abs(Hpr)
Upr = 1-abs(Upr)
Rpr = 1-abs(Rpr)
Bpr = 1-abs(Bpr)



fig, (ax1, ax2, ax3,ax4) = plt.subplots(1, 4, figsize=(12,9))
fig.suptitle('Pearson')
ax1.imshow(Hpr.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(Upr.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(Rpr.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(Bpr.reshape(mb1,nb1), cmap = plt.cm.gray)



Hpr = Scale(Hpr)
Upr = Scale(Upr)
Rpr = Scale(Rpr)
Bpr = Scale(Bpr)



MediaHpr = Binarize1(Hpr)
MediaHpr[MediaHpr>=1.0]=1.0
MediaHpr[MediaHpr<1.0]=0.0
MediaHpr = MediaHpr.astype(int)


MediaUpr =  Binarize1(Upr)
MediaUpr[MediaUpr>=1.0]=1.0
MediaUpr[MediaUpr<1.0]=0.0
MediaUpr = MediaUpr.astype(int)


MediaRpr =  Binarize1(Rpr)
MediaRpr[MediaRpr>=1.0]=1.0
MediaRpr[MediaRpr<1.0]=0.0
MediaRpr = MediaRpr.astype(int)

MediaBpr =  Binarize1(Bpr)
MediaBpr[MediaBpr>=1.0]=1.0
```

```
MediaBpr[MediaBpr<1.0]=0.0
MediaBpr = MediaBpr.astype(int)


#####

fig, (ax1, ax2, ax3,ax4) = plt.subplots(1, 4, figsize=(12,9))
fig.suptitle('Media Pearson')
ax1.imshow(MediaHpr.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(MediaUpr.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(MediaRpr.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(MediaBpr.reshape(mb1,nb1), cmap = plt.cm.gray)
```

Out[200]:

<matplotlib.image.AxesImage at 0x1b4df0fe948>



Media Pearson

In [201]:

```
OUT_H = (MediaEuH.reshape(m1,n1)+MediaHsc+MediaHsM+MediaHpr.reshape(m1,n1)+MediaH1norm.
OUT_U = (MediaEuU.reshape(mu1,nu1) +MediaUsM+MediaUsc+MediaUpr.reshape(mu1,nu1)+MediaU1
OUT_R = (MediaEuR.reshape(mr1,nr1) +MediaRsc+MediaRsM+MediaRpr.reshape(mr1,nr1)+MediaR1
OUT_B = (MediaEuB.reshape(mb1,nb1) +MediaBsc+MediaBsM+MediaBpr.reshape(mb1,nb1)+MediaB1
```

In [202]:

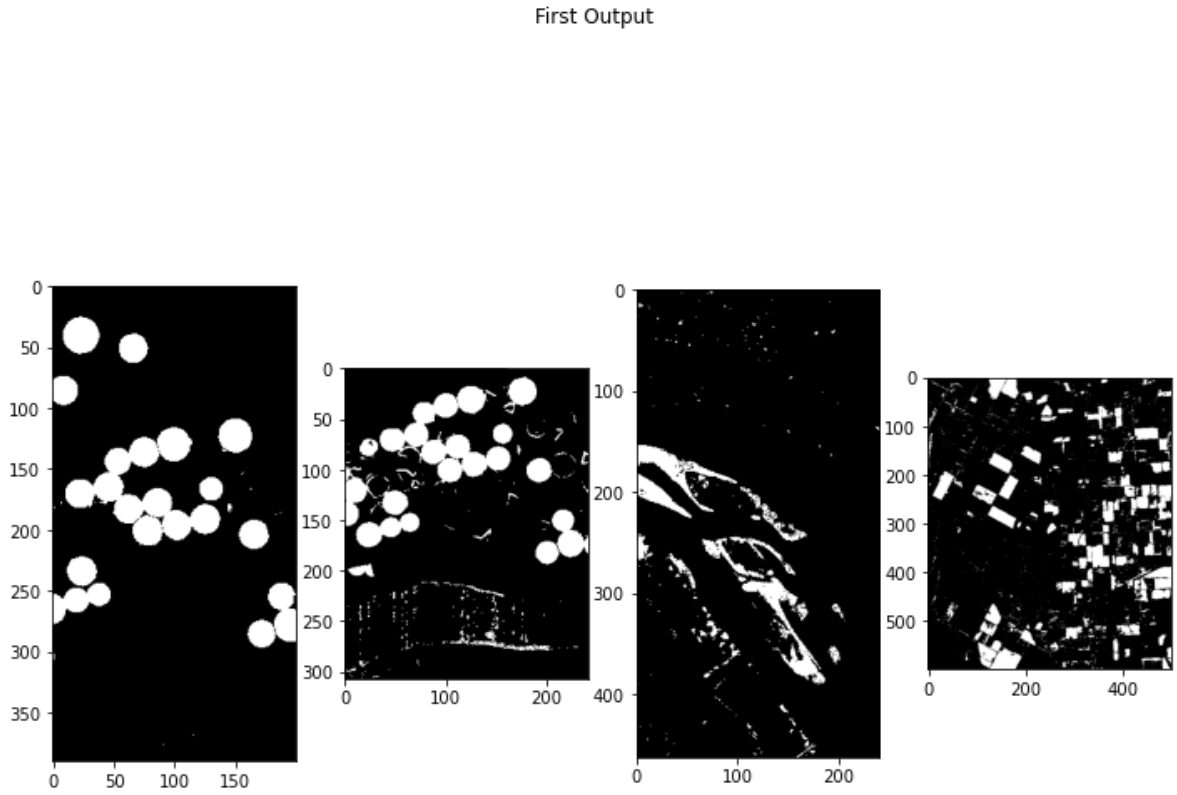
```
nOUT_H = np.copy(OUT_H)
nOUT_H[nOUT_H<3]=0
nOUT_H[nOUT_H>=3]=1


nOUT_U = np.copy(OUT_U)
nOUT_U[nOUT_U<3]=0
nOUT_U[nOUT_U>=3]=1


nOUT_R = np.copy(OUT_R)
nOUT_R[nOUT_R<3]=0
nOUT_R[nOUT_R>=3]=1

nOUT_B = np.copy(OUT_B)
nOUT_B[nOUT_B<3]=0
nOUT_B[nOUT_B>=3]=1
```

In [203]:

```
fig, (ax1, ax2, ax3,ax4) = plt.subplots(1, 4, figsize=(12,9))
fig.suptitle('First Output')
ax1.imshow(nOUT_H.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(nOUT_U.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(nOUT_R.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(nOUT_B.reshape(mb1,nb1), cmap = plt.cm.gray)
```

Out[203]:

```
<matplotlib.image.AxesImage at 0x1b4f07d6608>
```



Removing unclissified pixels from Bay_Area dataset

In [204]:

```
# Removing unclissified pixel location from ground-truth of Bay_Area dataset
ll = label_2.shape[0]
dim = mB*nB-ll
Mod_gt = np.zeros(dim)
partial = gtB.ravel()

j = 0
for i in range(mB*nB):
    if(partial[i]!= 2):
        Mod_gt[j] = partial[i]
        j = j+1
    else:
        continue


#Removing location of unclissified pixels from the generated output
Mod_out = np.zeros(dim)

Mod_mod = np.copy(nOUT_B)
Mod_mod[label_2[:,0], label_2[:,1]]=-1
Mod_mod = Mod_mod.ravel()

j = 0
for i in range(mB*nB):
    if(Mod_mod[i]!= -1):
        Mod_out[j] = Mod_mod[i]
        j = j+1
    else:
        continue
```

In [205]:

```
cnfH =confusion_matrix(new_gt, nOUT_H.ravel()).ravel()
OAH,KH = OA_K(cnfH)
print('\n overall accuracy Hermiston  ', OAH)
print('\n Kappa coeff Hermiston  ', KH)


cnfU =confusion_matrix(gtU.ravel(), nOUT_U.ravel()).ravel()
OAU,KU = OA_K(cnfU)
print('\n overall accuracy USA  ', OAU)
print('\n Kappa coeff USA  ', KU)


cnfR =confusion_matrix(gtR.ravel(), nOUT_R.ravel()).ravel()
OAR,KR = OA_K(cnfR)
print('\n overall accuracy River  ', OAR)
print('\n Kappa coeff River  ', KR)

cnfB =confusion_matrix(Mod_gt, Mod_out).ravel()
OAB,KB = OA_K(cnfB)
print('\n overall accuracy Bay  ', OAB)
print('\n Kappa coeff Bay  ', KB)

```

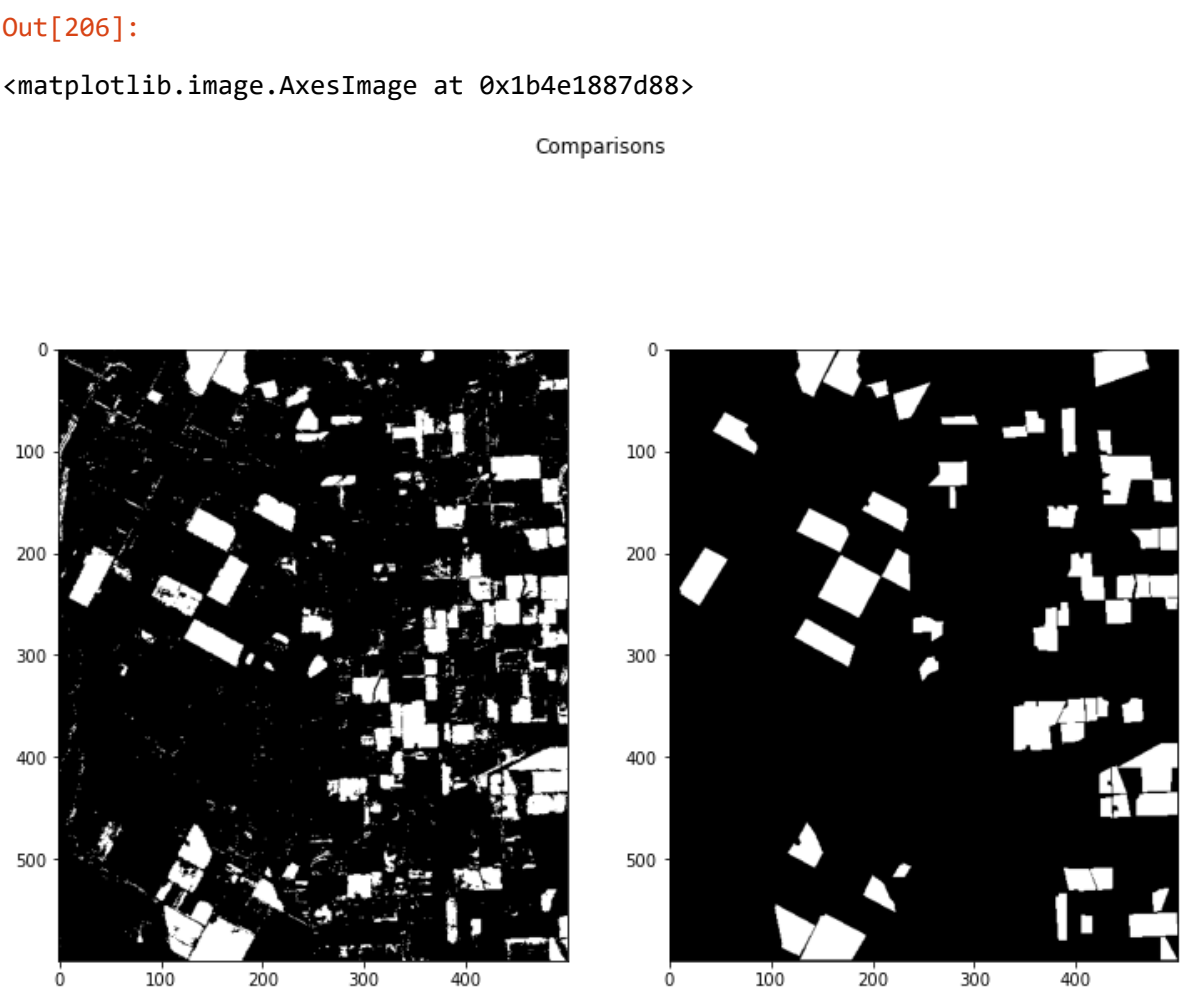
```
overall accuracy Hermiston   0.9866794871794872

Kappa coeff Hermiston   0.9410218110311578

overall accuracy USA   0.9076324219119575

Kappa coeff USA   0.7059968454086033

overall accuracy River   0.9624405151322334

Kappa coeff River   0.7250442980210057

overall accuracy Bay   0.8935222285121931

Kappa coeff Bay   0.5873311247140647
```

In [206]:

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,9))
fig.suptitle('Comparisons')
ax1.imshow(nOUT_B.reshape(mb1,nb1),cmap = plt.cm.gray)
ax2.imshow(NgtB.reshape(mb1,nb1), cmap = plt.cm.gray)
```

Out[206]:

```
<matplotlib.image.AxesImage at 0x1b4e1887d88>
```



Post-Processing Step:

- For 'Usa' and 'River' dataset, the obtained results can be improved by using the current output as pseudo-labels to a supervised binary classifier
- For 'Hermiston' dataset, since the produced output is a very neat binary map, it is convenient to adopt a pixel-wise cleaning/noise-removal method.

Difference images:

In [207]:

```
imgU = abs(U1-U2)
imgR = abs(R1-R2)
```

1st iteration Gaussian NB

In [208]:

```
clfU = GaussianNB()
outUF = clfU.fit(imgU, nOUT_U.reshape(mu1*nu1)).predict(imgU)

clfR = GaussianNB()
outRF = clfR.fit(imgR, nOUT_R.reshape(mr1*nr1)).predict(imgR)


cnfU =confusion_matrix(gtU.ravel(), outUF).ravel()
OAU,KU = OA_K(cnfU)
print('\n overall accuracy USA  ', OAU)
print('\n Kappa coeff USA  ', KU)


cnfR =confusion_matrix(gtR.ravel(), outRF).ravel()
OAR,KR = OA_K(cnfR)
print('\n overall accuracy River  ', OAR)
print('\n Kappa coeff River  ', KR)


```

```
 overall accuracy USA   0.953464797869896

 Kappa coeff USA   0.8609385737906876

 overall accuracy River   0.9609438713782565

 Kappa coeff River   0.7711179927663249
```

2nd iteration of Gaussian Nb

In [209]:

```
clfU = GaussianNB(var_smoothing=1e-2)
outUF = clfU.fit(imgU, outUF.reshape(mu1*nu1)).predict(imgU)
clfR = GaussianNB(var_smoothing=1.0)
outRF = clfR.fit(imgR, outRF.reshape(mr1*nr1)).predict(imgR)

cnfU =confusion_matrix(gtU.ravel(), outUF).ravel()
OAU,KU = OA_K(cnfU)
print('\n overall accuracy USA  ', OAU)
print('\n Kappa coeff USA  ', KU)


cnfR =confusion_matrix(gtR.ravel(), outRF).ravel()
OAR,KR = OA_K(cnfR)
print('\n overall accuracy River  ', OAR)
print('\n Kappa coeff River  ', KR)

```

```
 overall accuracy USA   0.9578980091097085

 Kappa coeff USA   0.8781070432396457

 overall accuracy River   0.9681223842341575

 Kappa coeff River   0.8013886129992429
```

For the Hermiston dataset instead it is convenient to remove the noise by adopting a pixel wise "filter". For such aim, a morphological opening transformation is chosen.

In [227]:

```
def clean_map1(change_map):
    change_map = change_map.astype(np.uint8)
    kernel1     = np.asarray(( 0,0,1,0,0),
                             (0,1,1,1,0),
                             (1,1,1,1,1),
                             (0,1,1,1,0),
                             (0,0,1,0,0)), dtype=np.uint8)

    cleanChangeMap = cv2.morphologyEx(change_map, cv2.MORPH_OPEN, kernel1)

    return cleanChangeMap
def clean_map2(change_map):
    change_map = change_map.astype(np.uint8)
    kernel1     = np.asarray((  0,1,1,0,1),
                             (0,0,1,1,1),
                             (1,0,1,0,0),
                             (1,0,1,1,1),
                             (1,1,1,1,1)), dtype=np.uint8)

    # With this ad-hoc kernel the accuracy is better, but the experiments are
    # done with standard kernel
    kernel     = np.asarray((  0,0,0,1,1,0,1),
                             (0,0,0,0,1,0,1),
                             (1,0,0,0,1,0,0),
                             (1,0,1,0,1,0,0),
                             (0,0,1,0,1,0,1),
                             (0,1,1,1,1,1,0),
                             (1,1,1,0,0,1,1)), dtype=np.uint8)

    cleanChangeMap = cv2.morphologyEx(change_map, cv2.MORPH_OPEN, kernel1)

    return cleanChangeMap
```

In [211]:

```
CleanedH = clean_map1(nOUT_H.reshape(m1,n1))
```

In [212]:

```
cnfH =confusion_matrix(new_gt, CleanedH.ravel()).ravel()
OAH,KH = OA_K(cnfH)
print('\n overall accuracy Hermiston  ', OAH)
print('\n Kappa coeff Hermiston  ', KH)
```

```
 overall accuracy Hermiston   0.9875512820512821

 Kappa coeff Hermiston   0.9446715027943061
```

In [228]:

```
CleanedB = clean_map2(nOUT_B.reshape(mb1,nb1))
cnfB =confusion_matrix(NgtB.ravel(), CleanedB.ravel()).ravel()
OAB,KB = OA_K(cnfB)
print('\n overall accuracy Bay  ', OAB)
print('\n Kappa coeff Bay  ', KB)
```

```
overall accuracy Bay   0.9290033333333333

Kappa coeff Bay   0.658672312506637
```

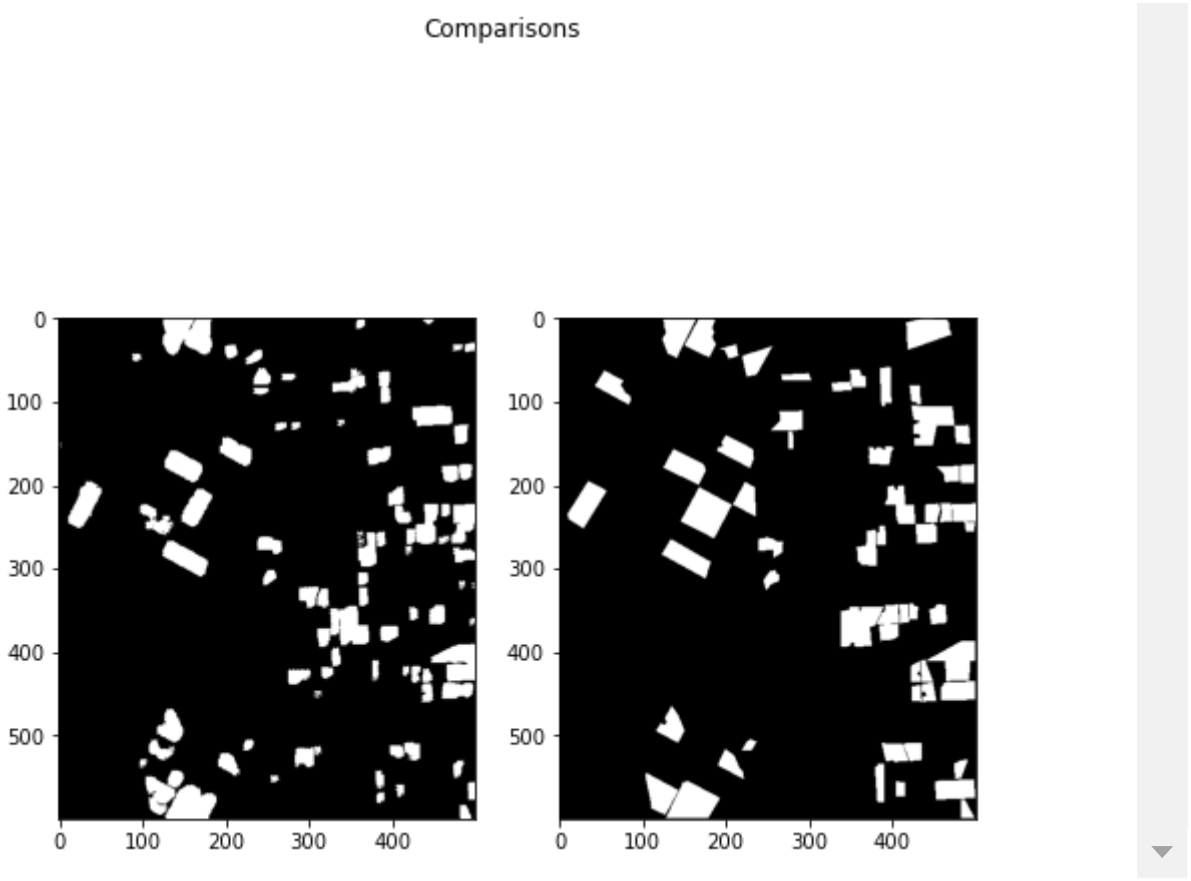
In [229]:

```
CleanedB.reshape(mb1,nb1)

#Removing location of unclissified pixels from the generated output
ll = label_2.shape[0]
dim = mb1*nb1-ll
Mod_clean = np.zeros(dim)

Mod_mod = (np.copy(CleanedB)).astype(int)

Mod_mod[label_2[:,0], label_2[:,1]]=-1
Mod_mod = Mod_mod.ravel()

j = 0
for i in range(mb1*nb1):
    if(Mod_mod[i]!= -1):
        Mod_clean[j] = Mod_mod[i]
        j = j+1
    else:
        continue



cnfB =confusion_matrix(Mod_gt, Mod_clean).ravel()
OAB,KB = OA_K(cnfB)
print('\n overall accuracy Bay  ', OAB)
print('\n Kappa coeff Bay  ', KB)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8.2,8))
fig.suptitle('Comparisons')
ax1.imshow(CleanedB.reshape(mb1,nb1),cmap = plt.cm.gray)
ax2.imshow(NgtB.reshape(mb1,nb1), cmap = plt.cm.gray)
```

```
 overall accuracy Bay   0.9217351987589969

 Kappa coeff Bay   0.6540428026820582
```

Out[229]:

```
<matplotlib.image.AxesImage at 0x1b4f4eeaf88>
```



In [215]:

```
fig, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4,figsize=(12,9))
ax1.imshow(CleanedH.reshape(m1,n1),cmap = plt.cm.gray)
ax2.imshow(outUF.reshape(mu1,nu1), cmap = plt.cm.gray)
ax3.imshow(outRF.reshape(mr1,nr1), cmap = plt.cm.gray)
ax4.imshow(CleanedB.reshape(mb1,nb1), cmap = plt.cm.gray)
plt.title('Final Output')

```

Out[215]:

```
Text(0.5, 1.0, 'Final Output')
```