Problem 10

Digital Image Processing

肖逸群 116033910112

Image representation and description

问题描述

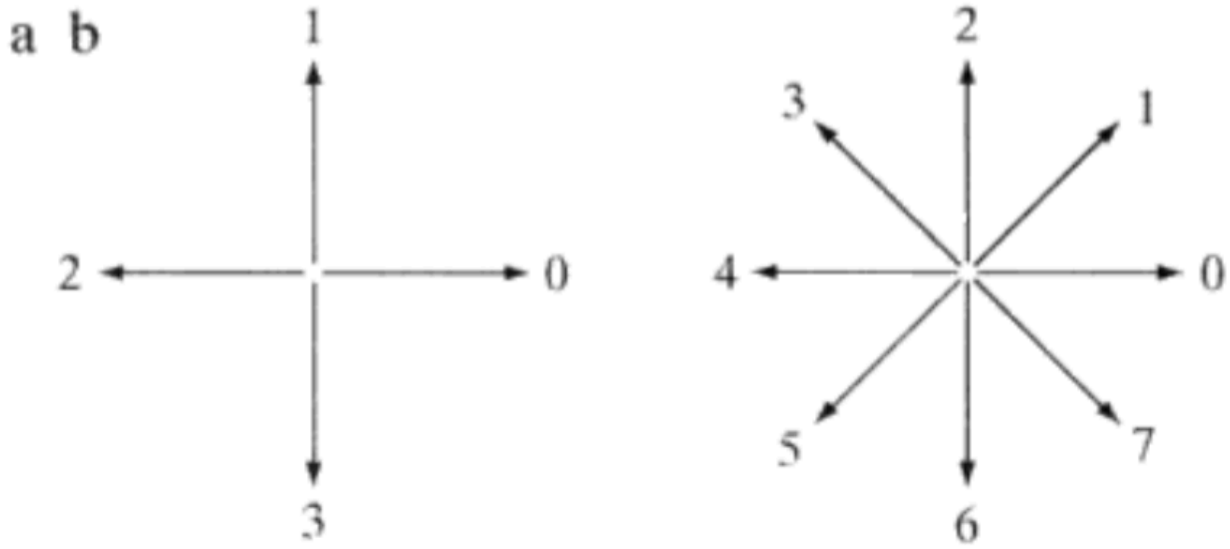
> (a). Develop a program to implement the boundary following algorithm, the resampling grid and calculate the chain code and the first difference chain code. Use the image 'noisy_stroke.tif' for test. (For technique details, please refer to pp.818-822 (3rd edition, Gonzalez DIP) or boundaryfollowing.pdf at the same address of the slides.)
> (b). Develop a program to implement the image description by the principal components (PC). Calculate and display the PC images and the reconstructed images from 2 PCs. Use the six images in 'washingtonDC.rar' as the test images.

算法思想

本题使用的均值掩模算法和 Ostu 阈值算法之前都有涉及，在此不再次进行原理解析。
本题的要求是实现边界追踪算法，边界追踪算法的步骤如下：

1. 令起始点 b_0 为图像中左上角标记为 1 的点，使用 c_0 表示 b_0 西侧的邻点。很明显，c_0 总是背景点。从 c_0 开始按顺时针方向考察 b_0 的 8 个邻点。令 b_1 表示所遇到的值为 1 的第一个邻点， 并直接令 c_1 (背景) 是序列中 b_1 之前的点。存储 b_0 和 b_1 的位置，以便在步骤 5 中使用。
2. 令 b = b_1 和 c = c_1。
3. 从 c 开始按顺时针方向进行，令 b 的 8 个邻点为 n_1, n_2, ..., n_8。找到标为 1 的第一个 nk。
4. 令 b = n*k* *和* *c* = *n*(k - 1)
5. 重复步骤 3 和步骤 4，直到 b = b_0 且找到的下一个边界点为 b_1。
   当算法停止时，所找到的 b 点的序列就构成了排列后的边界点的集合。
   实际上在本实验中，我把图像以阈值为 100 转化为灰度值为 0 与 255 的二值图像。

链码用于表示由顺次连接的具有指定长度和方向的直线段组成的边界。这种表示基于这些线段的4连接或者8连接。每个线段的方向使用一种数字编号方案编码，编码方式如下图所示：



边界的链码取决于起始点，然而，链码可以通过一个简单的过程关于起始点归一化，过程如下：
先简单地将链码视为方向号码的一个循环序列，并重新定义起始点，以便得到号码序列的最小整数值。此外，我们也可以针对旋转归一化，方法是使用链码的一次差分来替代链码本身，这个差分是通过计算链码中分隔两个相邻像素的方向变化的数得到的。

我们利用 PCA 来获取图像的主要成分，PCA 的步骤如下：

1. 对图像进行向量化，得到一个向量，对于图像中的某个位置，我使用该题目所涉及的 6 张图片来呈现该像素。
2. 计算协方差矩阵和特征值。
3. 使用霍特林变换来重构原始图像。

源码分析

- 实验过程如下：
    a. 读取原始图片，对其进行二值化操作，然后将其转为灰度图像。
    b. 对原始图像进行 9 * 9 规模的均值掩模操作，得到均值图像，达到去噪效果。
    c. 使用 Ostu 阈值算法来求出均值图像的边界。
    d. 使用边界追踪算法算出该图像的边界。
    e. 计算链码和差值码。

```
def main():
    originalImage = np.array(Image.open('./resource/noisy_stroke.tif'))
    m, n = originalImage.shape
    for i in range(m):
        for j in range(n):
            originalImage[i, j] = 255 if originalImage[i, j] > 100 else 0

    plt.subplot(2, 3, 1)
    plt.imshow(originalImage, cmap = plt.get_cmap('gray'))
    plt.title('Original')

    plt.subplot(2, 3, 2)
    l = 9
    averagingFilterImage = averagingFilter.averagingFilter(originalImage, l)
    plt.imshow(averagingFilterImage, cmap = plt.get_cmap('gray'))
    plt.title('%s * %s Averaging Mask' % (l, l))

    plt.subplot(2, 3, 3)
    ostuThresholdSegmentImage = thresholdSegmentAlgorithm.otsuThresholdSegment(averaging
```

```
FilterImage)
    plt.imshow(ostuThresholdSegmentImage, cmap = plt.get_cmap('gray'))
    plt.title('Ostu Threshold Segment')

    plt.subplot(2, 3, 4)
    boundaryImage, gridImage, gridConnectedImage = boundaryFollowingAlgorithm.boundaryFo
llowing(ostuThresholdSegmentImage)
    plt.imshow(boundaryImage, cmap = plt.get_cmap('gray'))
    plt.title('Boundary')

    plt.subplot(2, 3, 5)
    plt.imshow(gridImage, cmap = plt.get_cmap('gray'))
    plt.title('Grid Boundary')

    plt.subplot(2, 3, 6)
    plt.imshow(gridConnectedImage, cmap = plt.get_cmap('gray'))
    plt.title('Grid Connected Boundary')

    plt.show()

    chainCode, differentCode = boundaryFollowingAlgorithm.getCode(gridConnectedImage)
    print(chainCode)
    print(differentCode)
```

- 均值掩模算法和 Ostu 算法在第 2 和第 9 次作业有涉及，在此不再展示代码。
- 求取边界追踪的开始点

```
def getStartPoint(originalImage, s1, s2):
    m, n = originalImage.shape
    for i in range(s1, m):
        for j in range(s2, n):
            if originalImage[i, j] == 255:
                x = i
                y = j
                return x, y
```

- 求取边界追踪的下一个点

```
def getNextPoint(originalImage, bx_i, by_i, cx_i, cy_i):
    dx = [0, -1, -1, -1, 0, 1, 1, 1, 0, -1, -1, -1, 0, 1, 1, 1]
    dy = [1, 1, 0, -1, -1, -1, 0, 1, 1, 1, 0, -1, -1, -1, 0, 1]
    bx = bx_i
    by = by_i
    cx = cx_i
    cy = cy_i
    m, n = originalImage.shape
    for i in range(8):
        if bx + dx[i] == cx and by + dy[i] == cy:
            for j in range(i + 1, i + 8):
                tx = bx + dx[j]
                ty = by + dy[j]
                if tx >= 0 and tx < m and ty >= 0 and ty < n:
                    if originalImage[tx, ty] == 255:
                        bx = tx
                        by = ty
                        di = j
                        if di >= 8:
                            di -= 8
                        break
                cx = tx
```

```
            cy = ty
        break
    return bx, by, cx, cy, di
```

- 边界追踪函数

```
def boundaryFollowing(originalImage):
    m, n = originalImage.shape
    boundaryImage = np.zeros((m, n))

    startX, startY = getStartPoint(originalImage, 110, 220)
    bx = startX
    by = startY
    cx = startX
    cy = startY - 1
    b1x, b1y, c1x, c1y, discard = getNextPoint(originalImage, bx, by, cx, cy)

    while(1):
        boundaryImage[bx, by] = 255
        bx, by, cx, cy, discard = getNextPoint(originalImage, bx, by, cx, cy)
        if bx == startX and by == startY:
            b2x, b2y, c2x, c2y, discard = getNextPoint(originalImage, bx, by, cx, cy)
            if b1x == b2x and b1y == b2y:
                break

    density = 20
    gridImage = np.zeros((m, n))
    gridRows = np.floor(m / density)
    gridCols = np.floor(n / density)
    gridConnectedImage = np.zeros((gridRows, gridCols))

    for i in range(m):
        for j in range(n):
            if boundaryImage[i, j] == 255:
                gridX = np.floor(gridRows * (i + 1) / m)
                gridY = np.floor(gridCols * (j + 1) / n)
                bigGridX = np.floor(gridX / gridRows * m)
                bigGridY = np.floor(gridY / gridCols * n)
                gridConnectedImage[gridX, gridY] = 255
                gridImage[bigGridX, bigGridY] = 255

    return boundaryImage, gridImage, gridConnectedImage
```

- 计算链码和差值码函数

```
def getCode(gridImage):
    chainCode = []
    differentCode = []
    cnt = 0
    startX, startY = getStartPoint(gridImage, 10, 4)
    bx = startX
    by = startY
    cx = startX
    cy = startY - 1
    b1x, b1y, c1x, c1y, discard = getNextPoint(gridImage, bx, by, cx, cy)
    while 1:
        bx, by, cx, cy, di = getNextPoint(gridImage, bx, by, cx, cy)
        cnt += 1
        chainCode.append(di)
        if cnt >= 2:
            tmp = chainCode[cnt - 1] - chainCode[cnt - 2]
```

```
            if tmp < 0:
                tmp += 8
            differentCode.append(tmp)
        if bx == startX and by == startY:
            b2x, b2y, c2x, c2y, discard = getNextPoint(gridImage, bx, by, cx, cy)
            if b2x == b1x and b2y == b1y:
                break

    tmp = chainCode[0] - chainCode[cnt - 1]
    if tmp < 0:
        tmp += 8
    differentCode.append(tmp)

    return chainCode, differentCode
```

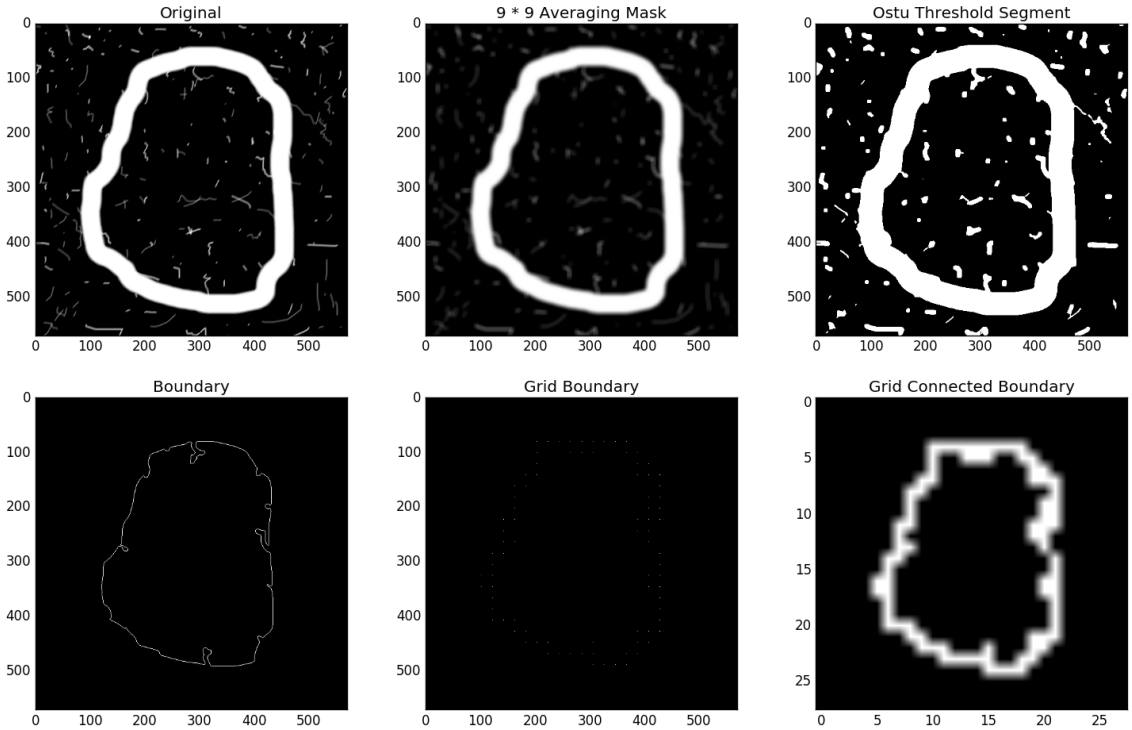
- PCA 函数

```
def principalComponentsTransform(originalImageSet):
    m, n = originalImageSet[0].size
    size = m * n
    vectors = np.zeros((size, 6))
    pixList = []
    for i in range(6):
        t = originalImageSet[i].load()
        pixList.append(t)
    ctr = 0
    for i in range(m):
        for j in range(n):
            for k in range(6):
                vectors[ctr, k] = pixList[k][i, j]
            ctr += 1

    mx = np.zeros((1, 6))
    for i in range(size):
        mx += vectors[i, :]
    mx = mx.T / (m * n)

    cx = np.zeros((6, 6))
    for i in range(size):
        cx += np.dot(vectors[i, :].T, vectors[i, :])
    cx /= size
    cx -= np.dot(mx, mx.T)
    eigenValue, eigenVector = np.linalg.eig(cx)
    vectorLen = len(eigenVector)
    eigenVectorT = eigenVector.copy()
    for i in range(vectorLen - 1, -1, -1):
        temp = eigenVector[i]
        eigenVectorT[vectorLen - 1 - i] = temp
    PCs = 2
    vectorMetas = np.zeros((size, PCs))
    for i in range(size):
        mxList = []
        for j in range(6):
            mxList.append(vectors[i, j] - mx[j])
        vectorMetas[i, :] = (np.dot(eigenVectorT[0 : PCs, :], mxList)).T

    vectorReconstructs = np.zeros((size, 6))
    for i in range(size):
        mxList = np.dot(eigenVectorT[0 : PCs, :].T, vectorMetas[i, :].T)
        for j in range(6):
            mxList[j] += mx[j]
```

```
        vectorReconstructs[i, :] = mxList.T

    reconstructsImageSet = []
    for i in range(6):
        temp = np.zeros((m, n))
        reconstructsImageSet.append(temp)
    ctr = 0
    for i in range(m):
        for j in range(n):
            for k in range(6):
                reconstructsImageSet[k][j, i] = vectorReconstructs[ctr, k]
            ctr += 1

    return reconstructsImageSet
```

实验结果

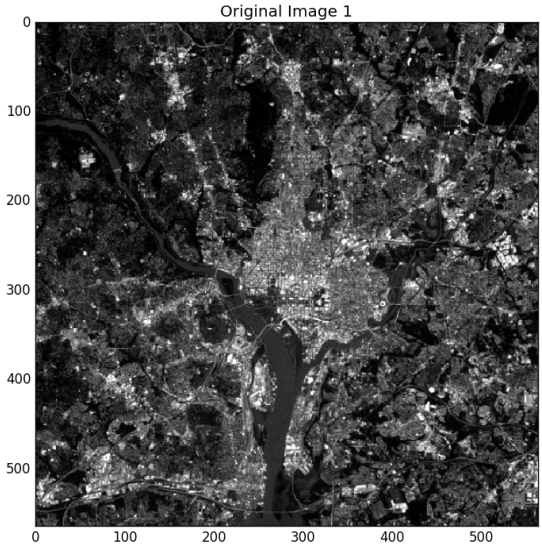
下图为对要求图像进行边界追踪的实验结果：

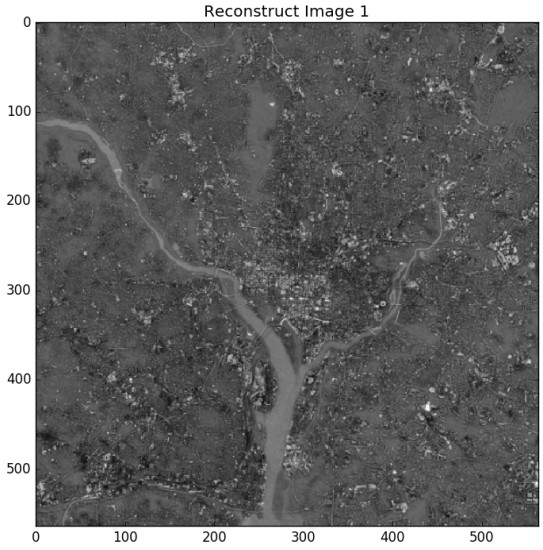


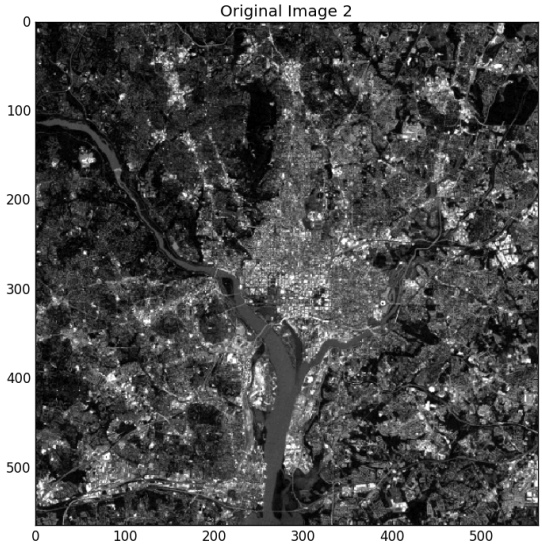
由于我发现使用 9 * 9 均值掩模的去噪效果并不是很理想，依旧有一些点被 Ostu 算法判定为边界点，故在进行边界追踪算法的时候选择从一个靠近中间区域的点开始，这样变可以找到正确的初始点了。
以下为链码和差值码的计算结果：

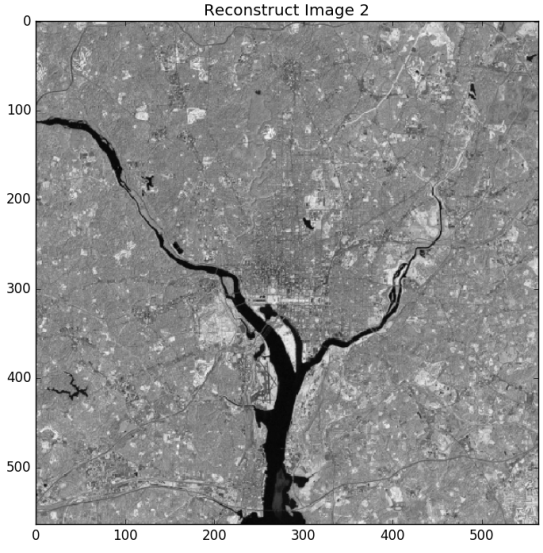
```
[5, 6, 6, 5, 6, 5, 6, 7, 6, 6, 0, 7, 7, 0, 7, 0, 0, 0, 7, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 1, 2, 2, 2, 2, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 6, 6, 5, 5, 6, 6]
[1, 0, 7, 1, 7, 1, 1, 7, 0, 2, 7, 0, 1, 7, 1, 0, 0, 7, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 7, 7, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 7, 0, 1, 0, 7]

Process finished with exit code 0
```

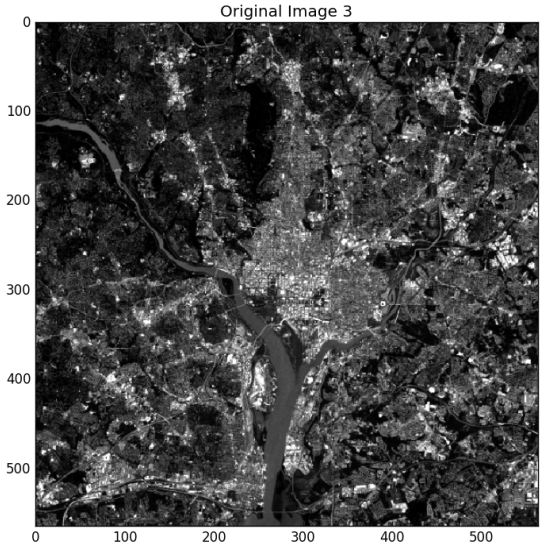
下列图片为对原始图像进行 PCA 处理的实验结果：

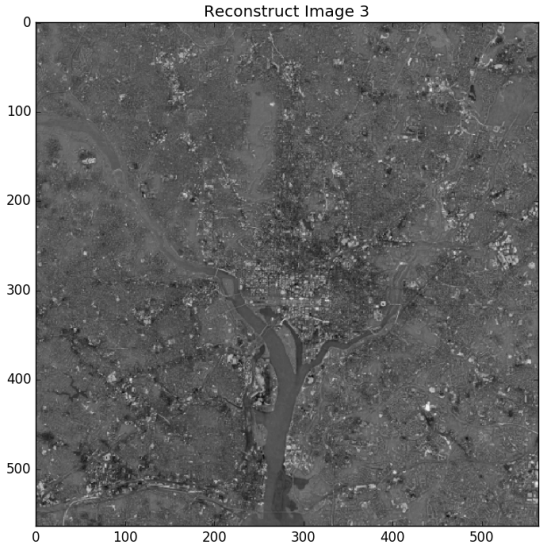

Original Image 1
0
100
200
300
400
500
0
100
200
300
400
500

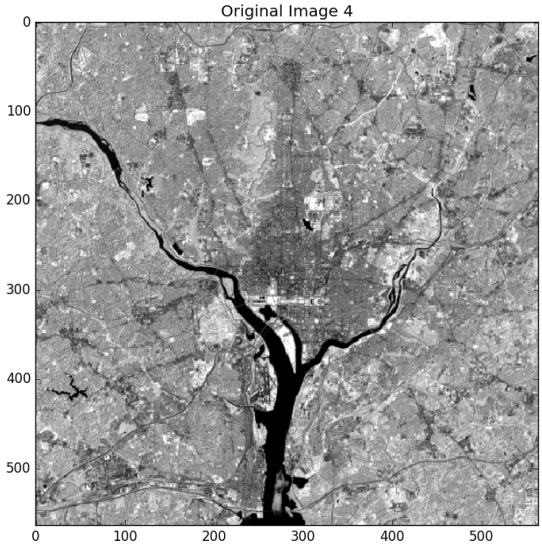

Reconstruct Image 1
0
100
200
300
400
500
0
100
200
300
400
500

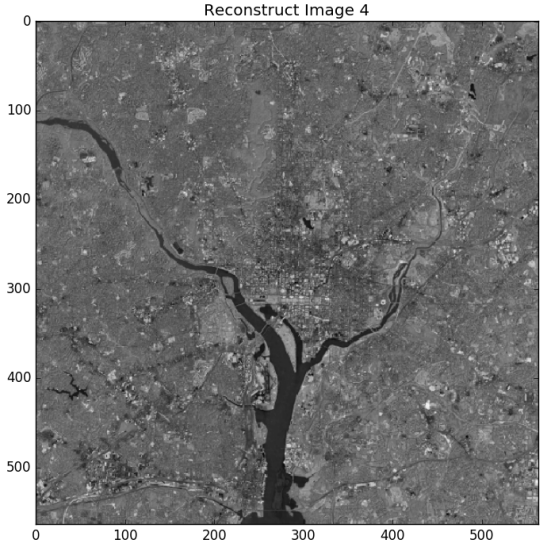

Original Image 2
0
100
200
300
400
500
0
100
200
300
400
500

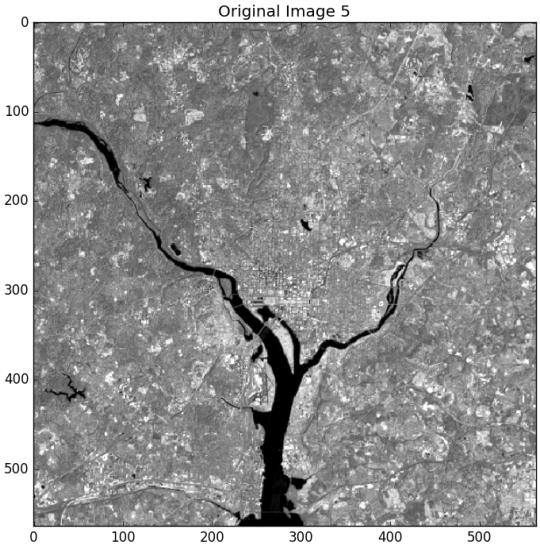

Reconstruct Image 2
0
100
200
300
400
500
0
100
200
300
400
500

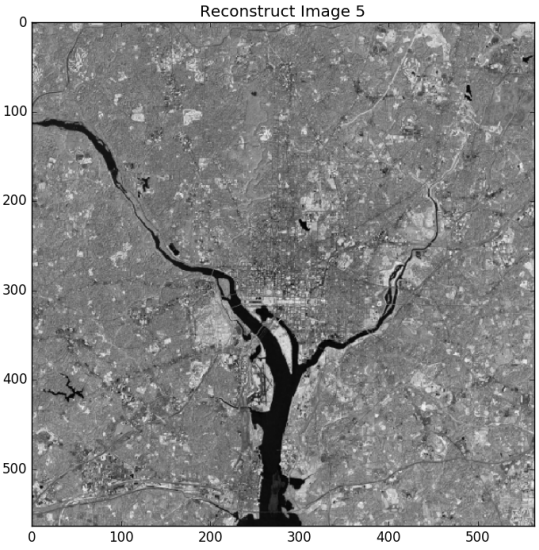

Original Image 3
0
100
200
300
400
500
0
100
200
300
400
500

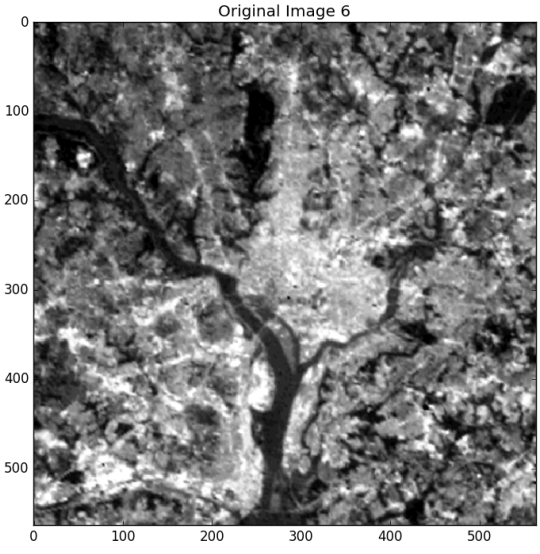

Reconstruct Image 3
0
100
200
300
400
500
0
100
200
300
400
500

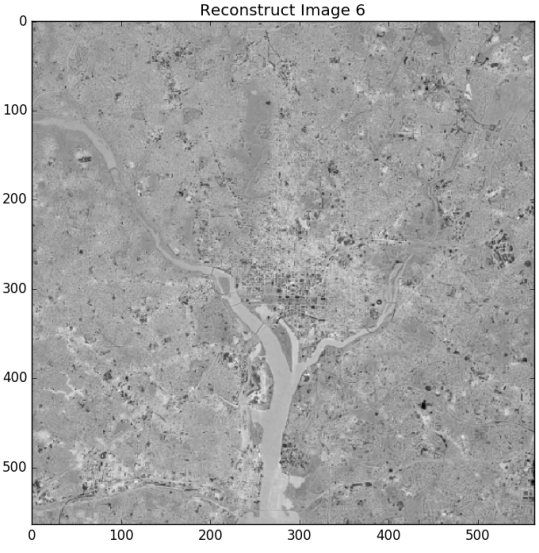

Original Image 4
0
100
200
300
400
500
0
100
200
300
400
500


Reconstruct Image 4
0
100
200
300
400
500
0
100
200
300
400
500


Original Image 5
0
100
200
300
400
500
0
100
200
300
400
500


Reconstruct Image 5
0
100
200
300
400
500
0
100
200
300
400
500


Original Image 6
0
100
200
300
400
500
0
100
200
300
400
500


Reconstruct Image 6
0
100
200
300
400
500
0
100
200
300
400
500