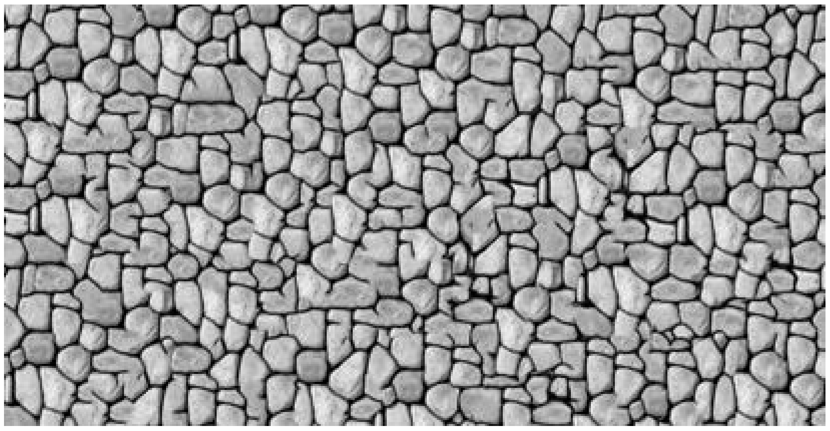


Texture Quilting (Due Saturday 2/23/2019)

In this assignment, you will develop code to stitch together image patches sampled from an input texture in order to synthesize new texture images. You can download the test image used to generate the example above from assignment folder Canvas.

You should start by reading through the whole assignment, looking at the provided code in detail to make sure you understand what it does. The main fucntion ***quilt_demo*** appears at the end. You will need to write several subroutines in order for it to function properly.

Name: Jeremy Taylor

SID: 14336420

---

1. Shortest Path [25 pts]

Write a function ***shortest_path*** that takes an 2D array of ***costs***, of shape HxW, as input and finds the *shortest vertical path* from top to bottom through the array. A vertical path is specified by a single horizontal location for each row of the H rows. Locations in successive rows should not differ by more than 1 so that at each step the path either goes straight or moves at most one pixel to the left or right. The cost is the sum of the costs of each entry the path traverses. Your function should return an length H vector that contains the index of the path location (values in the range 0..W-1) for each of the H rows.

You should solve the problem by implementing the dynamic programming algorithm described in class. You will have a for-loop over the rows of the "cost-to-go" array (M in the slides), computing the cost of the shortest path up to that row using the recursive formula that depends on the costs-to-go for the previous row. Once you have get to the last row, you can then find the smallest total cost. To find the path which actually has this smallest cost, you will need to do backtracking. The easiest way to do this is to also store the index of whichever minimum was selected at each location. These indices will also be an HxW array. You can then backtrack through these indices, reading out the path.

Finally, you should create at least three test cases by hand where you know the shortest path and see that the code gives the correct answer.

In [1]:
```
#modules used in your code
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```python
def shortest_path(costs):
    """
    This function takes an array of costs and finds a shortest path from the
    top to the bottom of the array which minimizes the total costs along the
    path. The path should not shift more than 1 location left or right between
    successive rows.

    In other words, the returned path is one that minimizes the total cost:

        total_cost = costs[0,path[0]] + costs[1,path[1]] + costs[2,path[2]] + ...

    subject to the constraint that:

        abs(path[i]-path[i+1])<=1

    Parameters
    ----------
    costs : 2D float array of shape HxW
         An array of cost values

    Returns
    -------
    path : 1D array of length H
        indices of a vertical path.  path[i] contains the column index of
        the path for each row i.
    """
    assert(np.issubdtype(costs.dtype, np.floating)), "costs must be a float array"
    assert(len(costs.shape)==2), "costs must be 2D"

    rows,cols = costs.shape

    # 2D array of min-cost path up to any given location, padded with inf to
    # keep logic for accessing neighbors consistent
    min_costs = np.pad(costs, (1,1), 'constant', constant_values=np.inf)

    # 2D array of indices such that a value at index i,j is the column of the smallest
    # neighbor in the row below it
    indices = np.zeros((rows,cols), dtype=int)
    indices[-1,:] = np.arange(cols)

    for i in range(rows-1, 0, -1):
        neighbors = np.stack(
            (min_costs[i+1,:-2], min_costs[i+1,1:-1], min_costs[i+1,2:]),
            axis=-1
        )
        min_costs[i,1:-1] = min_costs[i,1:-1] + np.min(neighbors,axis=1)
        indices[i-1,:] = np.argmin(neighbors,axis=1) - 1 + np.arange(cols)

    path = np.zeros(rows,dtype=int)

    path[0] = np.argmin(min_costs[1,:]) - 1

    for i in range(1,rows):
        path[i] = indices[i-1][path[i-1]]

    return path
```

```
In [3]: def enumerate_path(costs, path):
            """Return the values along the 1D shortest path through the 2D costs array"""
            return costs[np.arange(costs.shape[0]), path]

        # Test 1: shortest path straight through left most column
        costs1 = np.array(
            [
                [0.,10,11,12],
                [0.,7 ,8 ,9 ],
                [0.,4 ,5 ,6 ],
                [0.,1 ,2 ,3 ]
            ]
        )

        path1 = shortest_path(costs1)
        assert(np.array_equal(enumerate_path(costs1,path1), np.array([0.,0,0,0])))

        # Test 2: shortest path straight through right most column
        costs2 = np.array(
            [
                [20.,19,18, 17, 0],
                [16.,15,14, 13, 0],
                [12.,11,10, 9 , 0],
                [8. ,7 ,6 , 5 , 0],
                [4. ,3 ,2 , 1 , 0],
            ]
        )

        path2 = shortest_path(costs2)
        assert(np.array_equal(enumerate_path(costs2,path2), np.array([0.,0,0,0,0])))

        # Test 3: shortest path straight through middle column
        costs3 = np.array(
            [
                [1.,0,2],
                [1.,0,2],
                [1.,0,2],
                [1.,0,2],
            ]
        )

        path3 = shortest_path(costs3)
        assert(np.array_equal(enumerate_path(costs3,path3), np.array([0.,0,0,0])))

        # Test 4: diagonal path that crosses through middle
        costs4 = np.array(
            [
                [1., 1, 1, 1, 0],
                [1., 1, 1, 0, 1],
                [1., 1, 0, 1, 1],
                [1., 0, 1, 1, 1],
                [0., 1, 1, 1, 1],
            ]
        )

        path4 = shortest_path(costs4)
        assert(np.array_equal(enumerate_path(costs4,path4), np.array([0.,0,0,0,0])))

        # Test 5: diamond
```

```
costs5 = np.array(
    [
        [1., 1, 0, 1, 1],
        [1., 0, 1, 0, 1],
        [0., 1, 1, 1, 0],
        [1., 0, 1, 0, 1],
        [1., 1, 0, 1, 1],
    ]
)

path5 = shortest_path(costs5)
assert(np.array_equal(enumerate_path(costs5,path5), np.array([0.,0,0,0,0])))
print('passed all asserts\n')

# Test 6: random (verify manually)
print('random array test (verify manually):')

costs6 = np.random.randint(low=0,high=5, size=(3,3)).astype(np.float)
print('costs:\n',costs6)

path6 = shortest_path(costs6)
print('path:\n',path6)

print('path values:\n',enumerate_path(costs6,path6))
print('path cost:\n', np.sum(enumerate_path(costs6,path6)))
```

```
passed all asserts

random array test (verify manually):
costs:
 [[4. 0. 4.]
 [0. 4. 0.]
 [0. 3. 0.]]
path:
 [1 0 0]
path values:
 [0. 0. 0.]
path cost:
 0.0
```

2. Image Stitching: [25 pts]

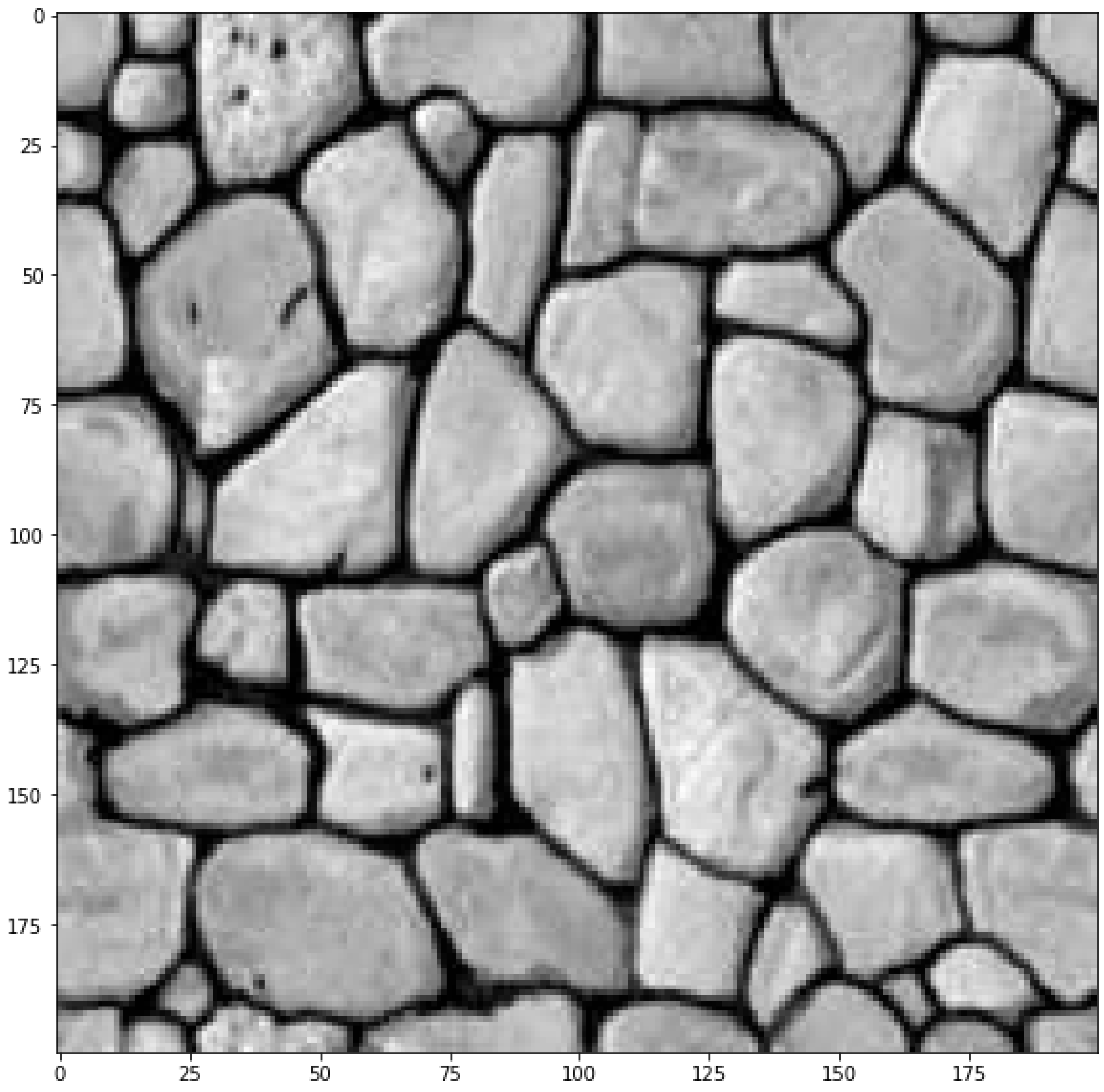
Write a function ***stitch*** that takes two gray-scale images, ***left_image*** and ***right_image*** and a specified ***overlap*** and returns a new output image by stitching them together along a seam where the two images have very similar brightness values. If the input images are of widths ***w1*** and ***w2*** then your stitched result image returned by the function should be of width ***w1+w2-overlap*** and have the same height as the two input images.

You will want to first extract the overlapping strips from the two input images and then compute a cost array given by the absolute value of their difference. You can then use your ***shortest_path*** function to find the seam along which to stitch the images where they differ the least in brightness. Finally you need to generate the output image by using pixels from the left image on the left side of the seam and from the right image on the right side of the seam. You may find it easiest to code this by first turning the path into an alpha mask for each image and then using the standard equation for compositing.

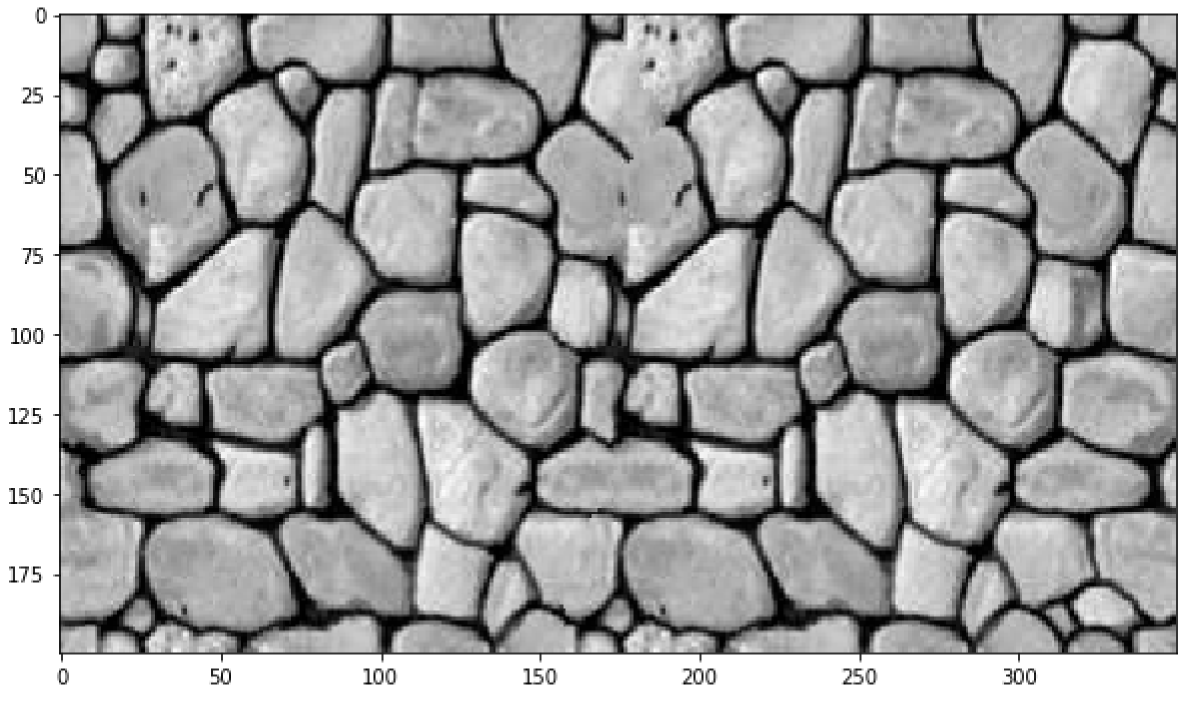
In [4]:
```python
def stitch(left_image, right_image, overlap):
    """
    This function takes a pair of images with a specified overlap and stitches them
    togther by finding a minimal cost seam in the overlap region.

    Parameters
    ----------
    left_image : 2D float array of shape HxW1
        Left image to stitch

    right_image : 2D float array of shape HxW2
        Right image to stitch

    overlap : int
        Width of the overlap zone between left and right image

    Returns
    -------
    stitched : 2D float array of shape Hx(W1+W2-overlap)
        The resulting stitched image
    """

    # inputs should be the same height
    assert(left_image.shape[0]==right_image.shape[0])

    left_overlap = left_image[:,left_image.shape[1]-overlap:]
    right_overlap = right_image[:,:overlap]

    cost = np.abs(right_overlap-left_overlap)
    path = shortest_path(cost)

    stitched = np.zeros(
        (left_image.shape[0], left_image.shape[1]+right_image.shape[1]-overlap)
    )

    for i in range(left_image.shape[0]):
        stitched[i,:(left_image.shape[1]-overlap+path[i])] = (
            left_image[i,:(left_image.shape[1]-overlap+path[i])])

        stitched[i,(left_image.shape[1]-overlap+path[i]):] = right_image[i,path[i]:]

    assert(stitched.shape[0]==left_image.shape[0])
    assert(stitched.shape[1]==(left_image.shape[1]+right_image.shape[1]-overlap))

    return stitched
```

```
In [5]: def imread_grayscale(path):
            I = plt.imread(path)
            if (I.dtype == np.uint8):
                I = I.astype(float) / 256
            if (len(I.shape) > 2):
                I = (I[:,:,0] + I[:,:,1] + I[:,:,2])/3
            return I

        I = imread_grayscale("rock_wall.jpg")

        plt.rcParams["figure.figsize"] = (10,10)
        print("original rock_wall.jpg image:")
        plt.imshow(I,cmap=plt.cm.gray)
        plt.show()

        stitched1 = stitch(I, I, 50)
        print("stitch(I,I,50):")
        plt.imshow(stitched1, cmap=plt.cm.gray)
        plt.show()

        stitched2 = stitch(I, np.flipud(np.fliplr(I)), 25)
        print("stitch(I, np.flipud(np.fliplr(I)), 25)")
        plt.imshow(stitched2, cmap=plt.cm.gray)
        plt.show()
```

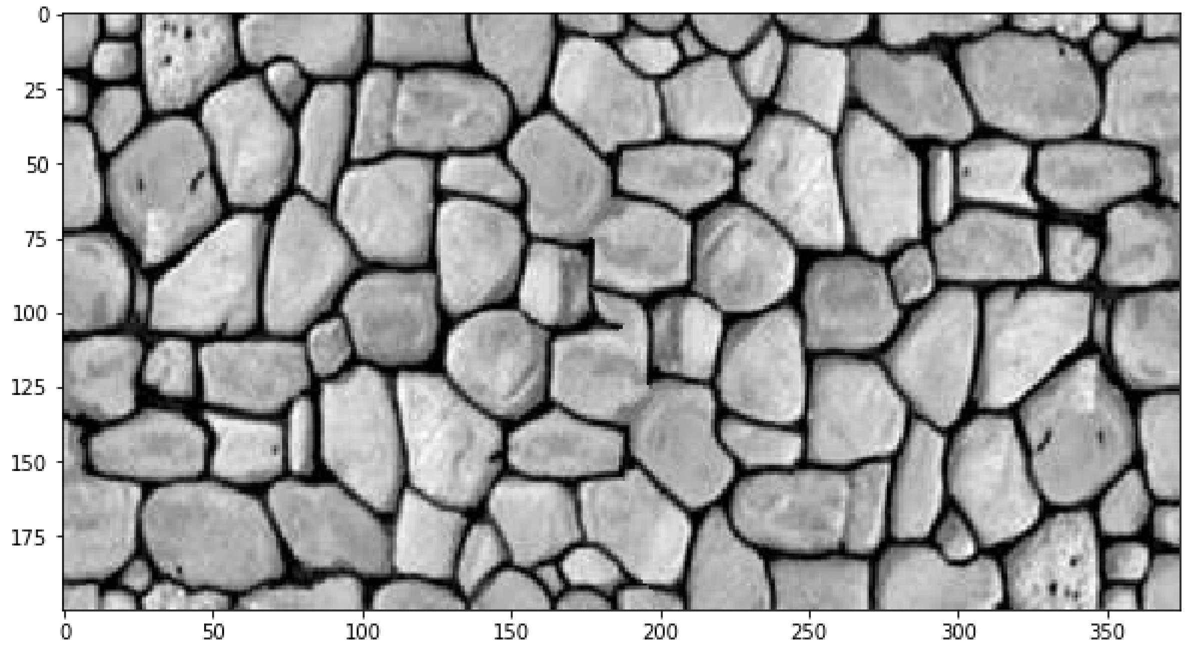
```
original rock_wall.jpg image:
```



```
stitch(I,I,50):
```



```
stitch(I, np.flipud(np.fliplr(I)), 25)
```



3. Texture Quilting: [25 pts]

Write a function ***synth_quilt*** that takes as input an array indicating the set of texture tiles to use, an array containing the set of available texture tiles, the ***tilesize*** and ***overlap*** parameters and synthesizes the output texture by stitching together the tiles. ***synth_quilt*** should utilize your stitch function repeatedly. First, for each horizontal row of tiles, construct the stitched row by repeatedly stitching the next tile in the row on to the right side of your row image. Once you have row images for all the rows, you can stitch them together to get the final image. Since your stitch function only works for vertical seams, you will want to transpose the rows, stitch them together, and then transpose the result. You may find it useful to look at the provided code below which simply puts down the tiles with the specified overlap but doesn't do stitching. Your quilting function will return a similar result but with much smoother transitions between the tiles.

```
In [6]: def synth_quilt(tile_map,tiledb,tilesize,overlap):

            """
            This function takes the name of an image and quilting parameters and synthesizes a
            new texture image by stitching together sampled tiles from the source image.


            Parameters
            ----------
            tile_map : 2D array of int
                Array storing the indices of which tiles to paste down at each output location

            tiledb : list of int
                Dimensions of output in tiles,  e.g. (3,4)

            tilesize : (int,int)
                Size of a tile in pixels

            overlap : int
                Amount of overlap between tiles

            Returns
            -------
            output : 2D float array
                The resulting synthesized texture of size
            """

            # determine output size based on overlap and tile size
            outh = (tilesize[0]-overlap)*tile_map.shape[0] + overlap
            outw = (tilesize[1]-overlap)*tile_map.shape[1] + overlap
            output = np.zeros((outh,outw))

            def get_tile_image(i,j):
                tile_vec = tiledb[tile_map[i,j],:]
                return np.reshape(tile_vec,tilesize)

            for i in range(tile_map.shape[0]):
                icoord = i*(tilesize[0]-overlap)

                # create local row_image to separate output from current row of tiles
                row_image = np.zeros((tilesize[0], outw))

                # paste the first tile in-place in the row
                row_image[:, 0:tilesize[1]] = get_tile_image(i,0)

                for j in range(1,tile_map.shape[1]):
                    jcoord_left = (j-1)*(tilesize[1]-overlap)
                    jcoord = j*(tilesize[1]-overlap)

                    # get the current tile image
                    tile_image = get_tile_image(i,j)

                    # progress on the row image up until tile i,j
                    row_image_left = row_image[:, 0:(tilesize[1]+jcoord_left)]

                    # stitch tile i,j onto the end of the row progress
                    row_image[:,0:jcoord+tilesize[1]] = (
                        stitch(row_image_left, tile_image, overlap))
```

```
        # stitch together row images
        if i == 0:
            # paste first row down in-place
            output[0:tilesize[0],:] = row_image
        else:
            # stitch row i>=1 to the previous row
            icoord_up = (i-1)*(tilesize[0]-overlap)

            # progress on the image up until this row
            row_image_up = output[0:(tilesize[0]+icoord_up),:].transpose()

            # stitch rows together
            output[0:(icoord+tilesize[0]), :] = (
                stitch(row_image_up, row_image.transpose(), overlap).transpose())

    return output
```

4. Texture Synthesis Demo [25pts]

The function provided below ***quilt_demo*** puts together the pieces. It takes a sample texture image and a specified output size and uses the functions you've implemented previously to synthesize a new texture sample.

You should write some additional code in the cells that follow to in order demonstrate the final result and experiment with the algorithm parameters in order to produce a compelling visual result and write explanations of what you discovered.

Test your code on the provided image *rock_wall.jpg*. There are three parameters of the algorithm. The *tilesize*, *overlap* and *K. In the provided* *texture_demo*** code below, these have been set at some default values. Include in your demo below images of three example texture outputs when you: (1) increase the tile size, (2) decrease the overlap, and (3) decrease the value for K. For each result explain how it differs from the default setting of the parameters and why.

Test your code on two other texture source images of your choice. You can use images from the web or take a picture of a texture yourself. You may need to resize or crop your input image to make sure that the ***tiledb*** is not overly large. You will also likely need to modify the ***tilesize*** and ***overlap*** parameters depending on your choice of texture. Once you have found good settings for these parameters, synthesize a nice output texture. Make sure you display both the image of the input sample and the output synthesis for your two other example textures in your submitted pdf.

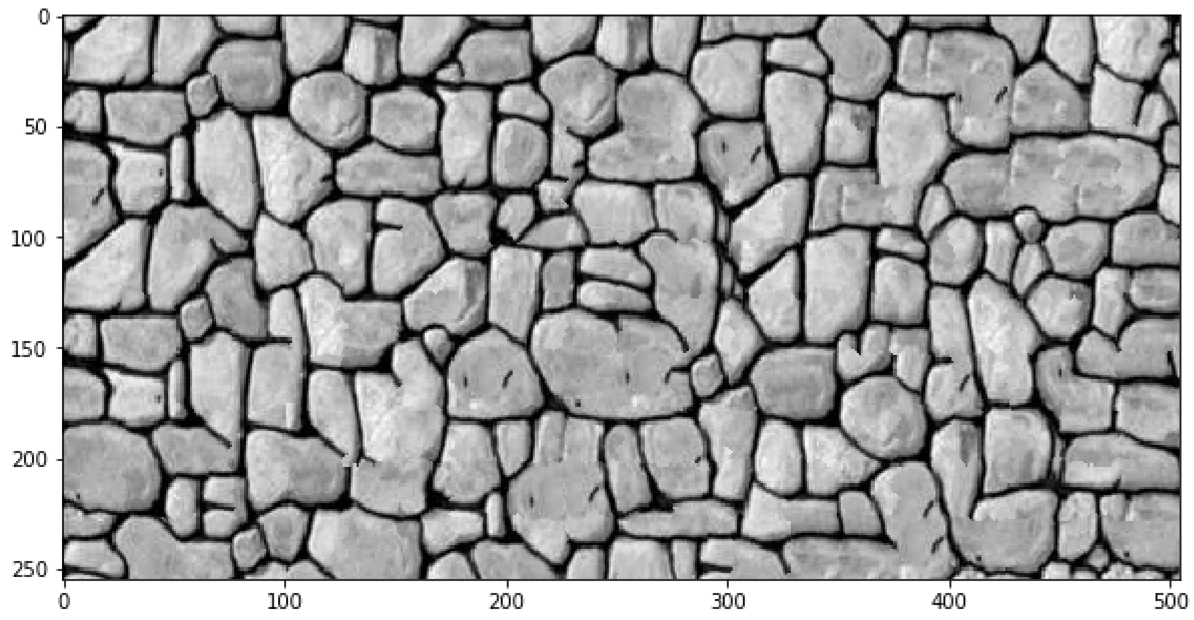
In [7]:
```python
#skimage is only needed for sample tiles code provided below
#you should not use it in your own code
import skimage as ski

def sample_tiles(image,tilesize,randomize=True):
    """
    This function generates a library of tiles of a specified size from a given source i

    Parameters
    ----------
    image : float array of shape HxW
        Input image

    tilesize : (int,int)
        Dimensions of the tiles in pixels


    Returns
    -------
    tiles : float array of shape  npixels x numtiles
        The library of tiles stored as vectors where npixels is the
        product of the tile height and width
    """

    tiles = ski.util.view_as_windows(image,tilesize)
    ntiles = tiles.shape[0]*tiles.shape[1]
    npix = tiles.shape[2]*tiles.shape[3]
    assert(npix==tilesize[0]*tilesize[1])

    print("library has ntiles = ",ntiles,"each with npix = ",npix)

    tiles = tiles.reshape((ntiles,npix))

    # randomize tile order
    if randomize:
        tiles = tiles[np.random.permutation(ntiles),:]

    return tiles


def topkmatch(tilestrip,dbstrips,k):
    """
    This function finds the top k candidate matches in dbstrips that
    are most similar to the provided tile strip.

    Parameters
    ----------
    tilestrip : 1D float array of length npixels
        Grayscale values of the query strip

    dbstrips : 2D float array of size npixels x numtiles
        Array containing brightness values of numtiles strips in the database
        to match to the npixels brightness values in tilestrip

    k : int
        Number of top candidate matches to sample from

    Returns
    -------
    matches : list of ints of length k
        The indices of the k top matching tiles
    """
    assert(k>0)
    assert(dbstrips.shape[0]>k)
    error = (dbstrips-tilestrip)
    ssd = np.sum(error*error,axis=1)
    ind = np.argsort(ssd)
    matches = ind[0:k]
    return matches


def quilt_demo(sample_image, ntilesout=(10,20), tilesize=(30,30), overlap=5, k=5):
    """
    This function takes the name of an image

    Parameters
    ----------
    sample_image : 2D float array
        Grayscale image containing sample texture

    ntilesout : list of int
        Dimensions of output in tiles,  e.g. (3,4)

    tilesize : int
        Size of the square tile in pixels

    overlap : int
        Amount of overlap between tiles

    k : int
        Number of top candidate matches to sample from

    Returns
    -------
    img : list of int of length K
        The resulting synthesized texture of size
    """

    # generate database of tiles from sample
    tiledb = sample_tiles(sample_image,tilesize)
    # number of tiles in the database
    nsampletiles = tiledb.shape[0]

    if (nsampletiles<k):
        print("Error: tile database is not big enough!")

    # generate indices of the different tile strips
    i,j = np.mgrid[0:tilesize[0],0:tilesize[1]]
    top_ind = np.ravel_multi_index(np.where(i<overlap),tilesize)
    bottom_ind = np.ravel_multi_index(np.where(i>=tilesize[0]-overlap),tilesize)
    left_ind = np.ravel_multi_index(np.where(j<overlap),tilesize)
    right_ind = np.ravel_multi_index(np.where(j>=tilesize[1]-overlap),tilesize)

    # initialize an array to store which tile will be placed
    # in each location in the output image
    tile_map = np.zeros(ntilesout,'int')

    #print('row:')
    for i in range(ntilesout[0]):
        #print(i)
        for j in range(ntilesout[1]):

            if (i==0)&(j==0):                 # first row first tile
                matches = np.zeros(k) #range(nsampletiles)

            elif (i==0):                      # first row (but not first tile)
                left_tile = tile_map[i,j-1]
                tilestrip = tiledb[left_tile,right_ind]
                dbstrips = tiledb[:,left_ind]
                matches = topkmatch(tilestrip,dbstrips,k)

            elif (j==0):                      # first column (but not first row)
                above_tile = tile_map[i-1,j]
                tilestrip = tiledb[above_tile,bottom_ind]
                dbstrips = tiledb[:,top_ind]
                matches = topkmatch(tilestrip,dbstrips,k)

            else:                             # neigbors above and to the left
                left_tile = tile_map[i,j-1]
                tilestrip_1 = tiledb[left_tile,right_ind]
                dbstrips_1 = tiledb[:,left_ind]
                above_tile = tile_map[i-1,j]
                tilestrip_2 = tiledb[above_tile,bottom_ind]
                dbstrips_2 = tiledb[:,top_ind]
                # concatenate the two strips
                tilestrip = np.concatenate((tilestrip_1,tilestrip_2))
                dbstrips = np.concatenate((dbstrips_1,dbstrips_2),axis=1)
                matches = topkmatch(tilestrip,dbstrips,k)

            #choose one of the k matches at random
            tile_map[i,j] = matches[np.random.randint(0,k)]

    output = synth_quilt(tile_map,tiledb,tilesize,overlap)

    return output
```

```
In [26]: # load in rock_wall.jpg
         # run and display results for quilt_demo with
         #

         # (rock_wall.jpg was loaded earlier as I)

         # (0) default parameters
         quilt0 = quilt_demo(I)

         # (1) increased tile size
         quilt1 = quilt_demo(I,tilesize=(50,50))

         # (2) decrease the overlap
         quilt2 = quilt_demo(I,overlap=1)

         # (3) increase the value for K.
         quilt3 = quilt_demo(I,k=800)
```

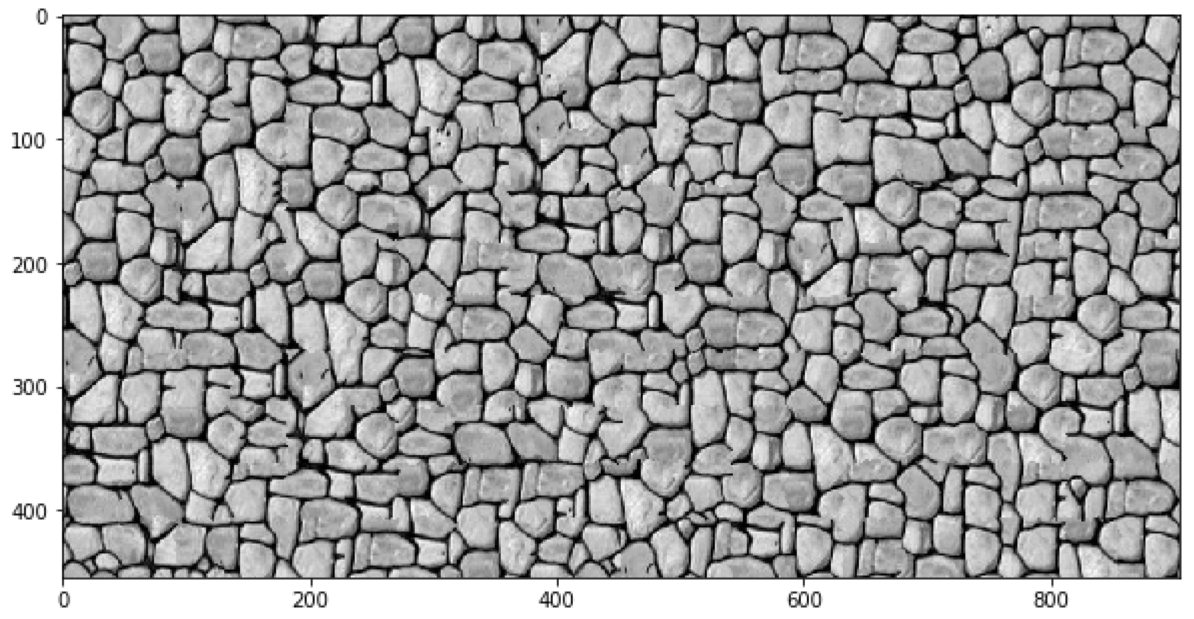
```
library has ntiles =  29241 each with npix =  900
library has ntiles =  22801 each with npix =  2500
library has ntiles =  29241 each with npix =  900
library has ntiles =  29241 each with npix =  900
```

```
In [27]: plt.rcParams["figure.figsize"] = (10,10)
         print("default parameters:")
         plt.imshow(quilt0, cmap=plt.cm.gray)
         plt.show()

         print("increased tile size to (50,50)")
         plt.imshow(quilt1, cmap=plt.cm.gray)
         plt.show()

         print("decreased overlap to 1")
         plt.imshow(quilt2, cmap=plt.cm.gray)
         plt.show()

         print("increased K to 800")
         plt.imshow(quilt3, cmap=plt.cm.gray)
         plt.show()
```

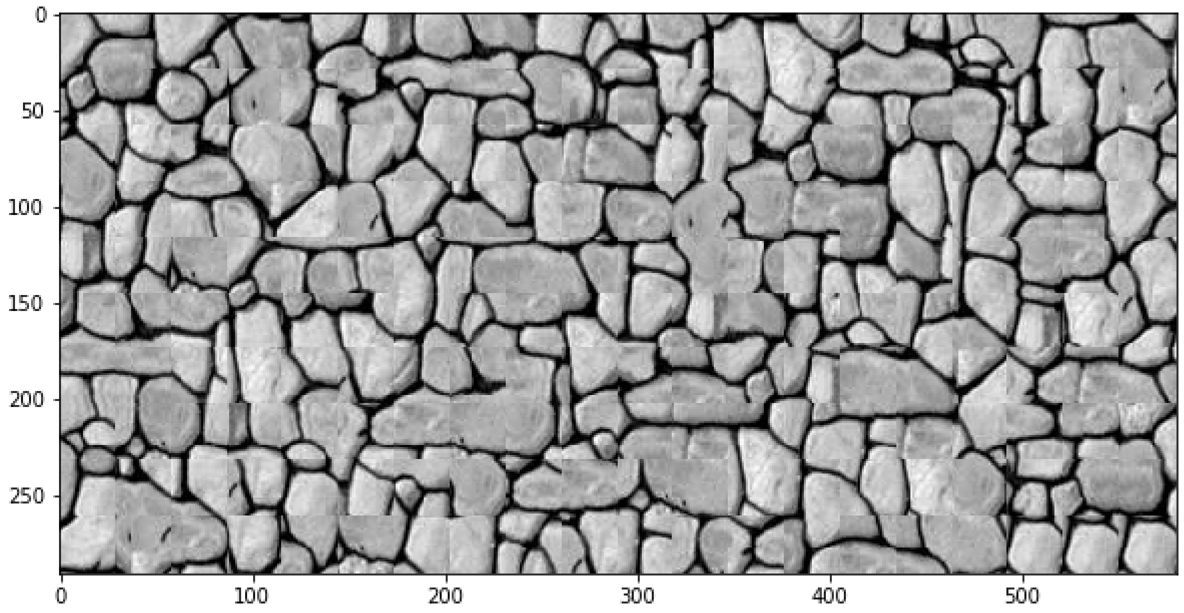
```
default parameters:
```



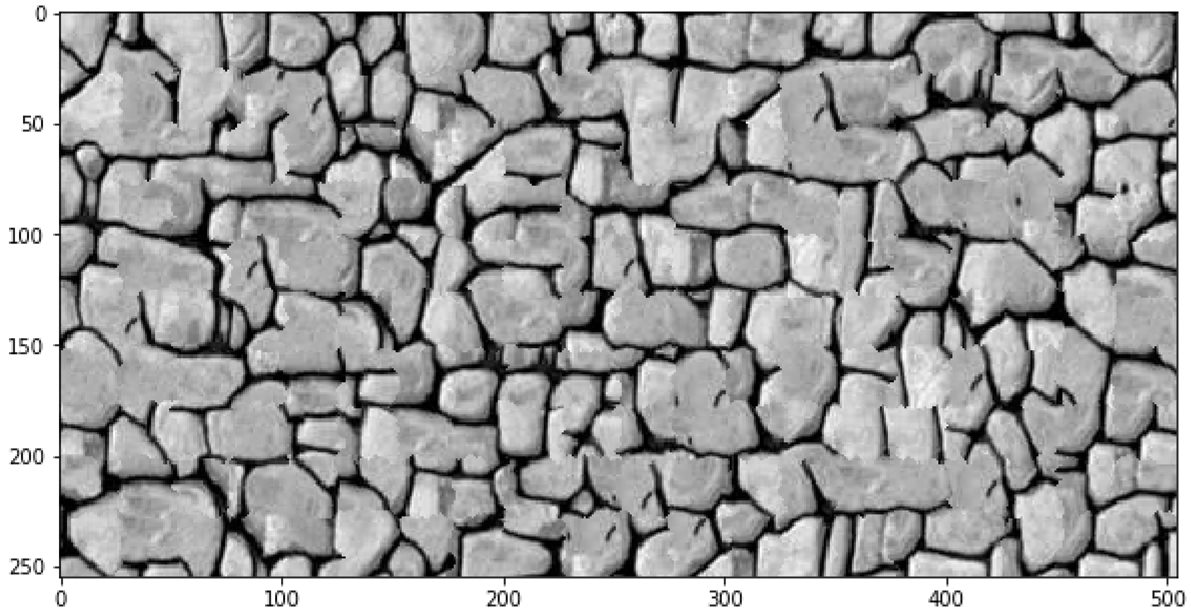
```
increased tile size to (50,50)
```



```
decreased overlap to 1
```



```
increased K to 800
```



For each result shown, explain here how it differs visually from the default setting of the parameters and explain why:

(1) Increasing tile size

The tile size changes the size of the sampling window over the input image; this affects the "features" of the input image that get selected and also the size of the output quilt. For the rock wall image, this means more individual stones are used in the texture quilt, and the size of each stone relative to the quilt seems smaller.

(2) Decreasing overlap

Decreasing the overlap causes the stitching function to have fewer seam paths to consider. If the overlap is reduced to 1 pixel, there is only one path possible but it may not be the best local min-cost path. Thus, small overlap values may result in abrupt changes in brightness values along the seams between tiles and rows.

(3) Increasing K

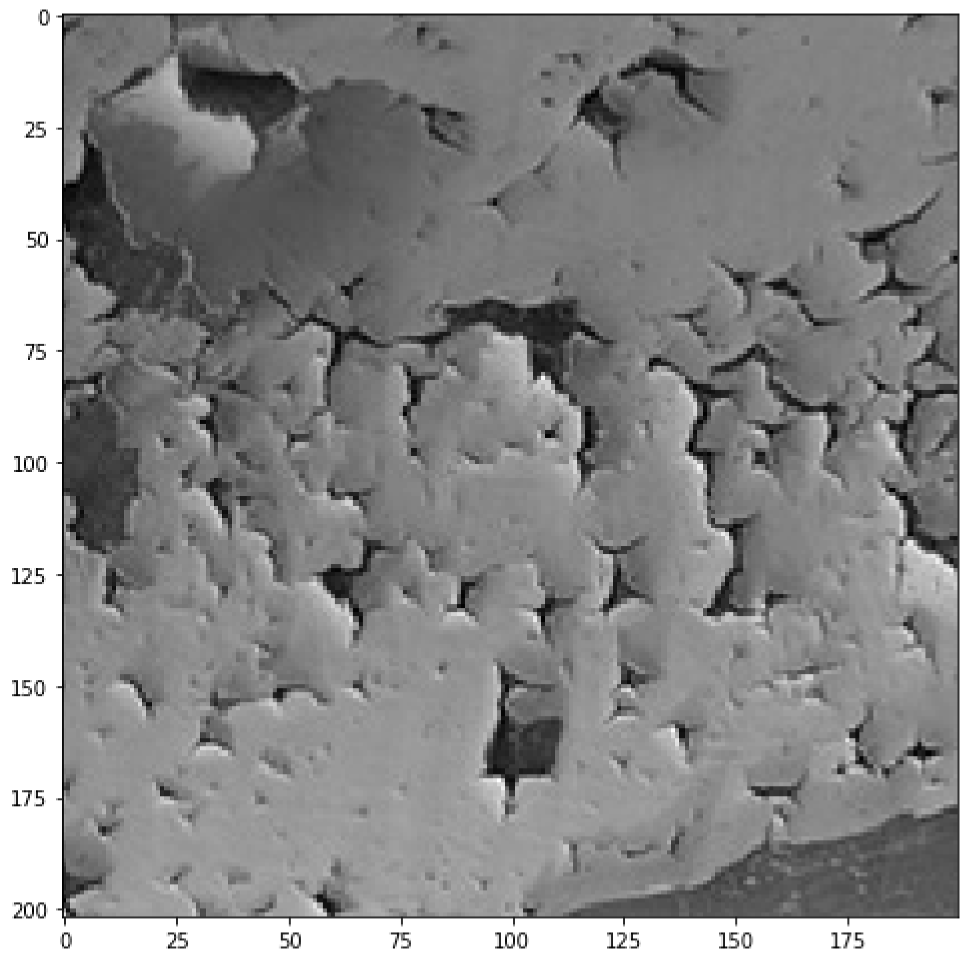
The value of K controls the number of similar tiles to consider when picking tiles for the tile_map. Because the similar tiles are sorted in order of decreasing similarity, and the tile chosen out of the K is random, increasing K will decrease the probability that neighboring tiles are similar. A large value of K may create sharper seams where neighboring tiles have very differing brightness values, and the chosen seam is still relatively costly.

In [22]:
```
img1 = imread_grayscale("cracked_paint.jpg")
img1_quilt = quilt_demo(img1, ntilesout=(15,15), overlap=10, tilesize=(35,35),k=4)
```

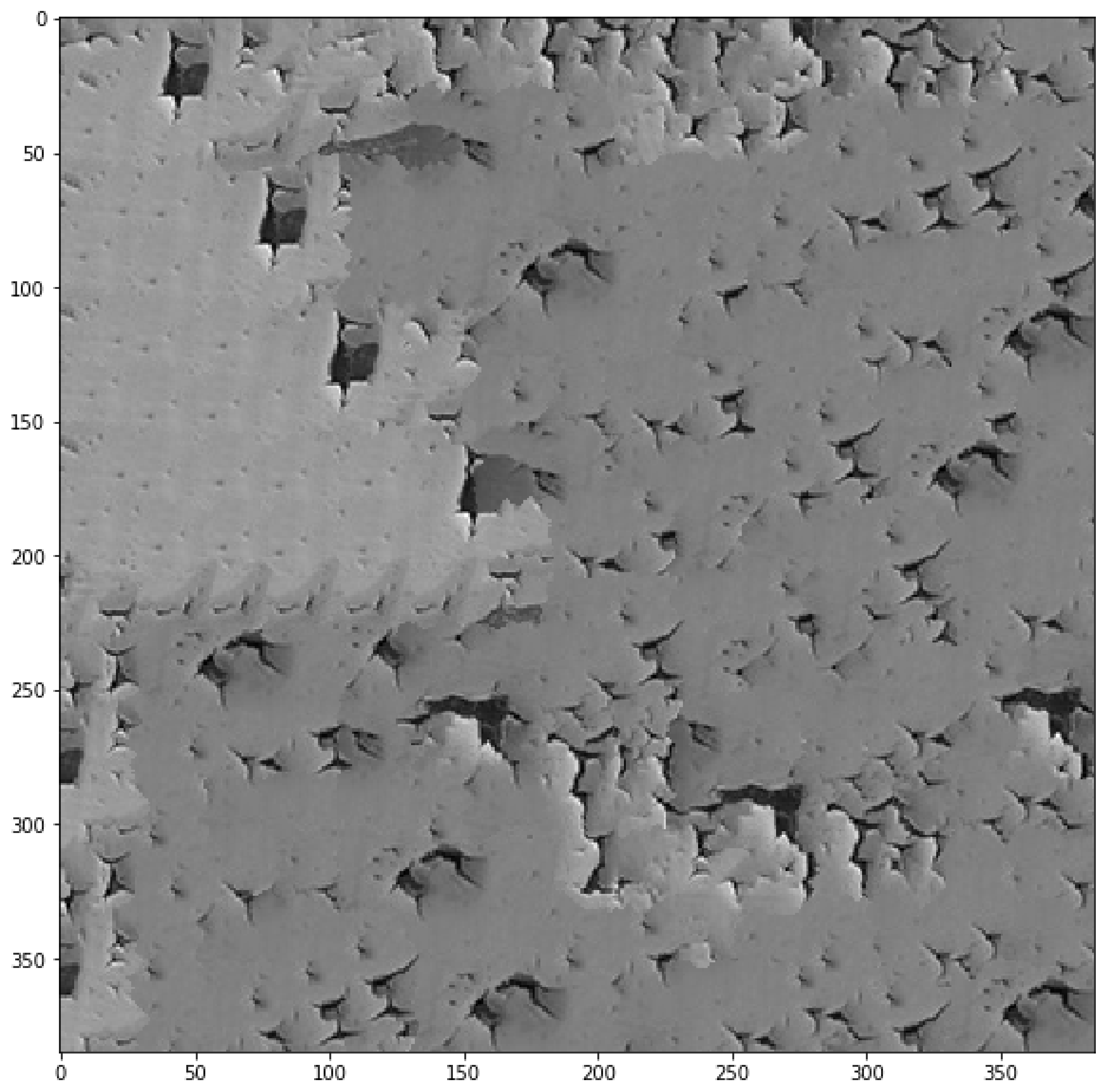
```
library has ntiles =  27888 each with npix =  1225
```

In [28]:
```
plt.rcParams["figure.figsize"] = (8,10)
print("original image: ")
plt.imshow(img1, cmap=plt.cm.gray)
plt.show()

plt.rcParams["figure.figsize"] = (15,10)
print("quilt: ")
plt.imshow(img1_quilt, cmap=plt.cm.gray)
plt.show()
```

```
original image:
```



```
quilt:
```

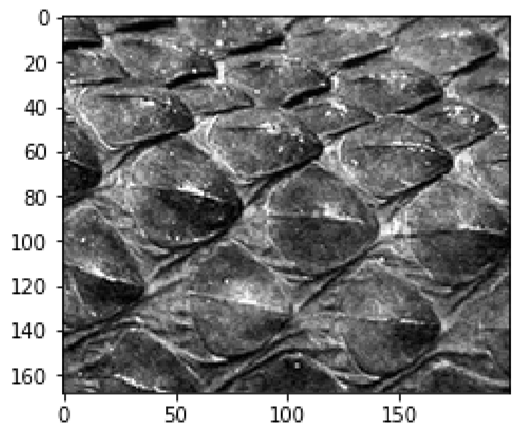


In [24]:
```python
img2 = imread_grayscale("scales.jpg")
img2_quilt = quilt_demo(img2, ntilesout=(25,15),tilesize=(25,40), overlap=8)
```

```
library has ntiles =  23184 each with npix =  1000
```

In [29]:
```
plt.rcParams["figure.figsize"] = (4,8)
print("original image: ")
plt.imshow(img2, cmap=plt.cm.gray)
plt.show()

plt.rcParams["figure.figsize"] = (12,8)
print("quilt: ")
plt.imshow(img2_quilt, cmap=plt.cm.gray)
plt.show()
```

```
original image:
```



```
quilt:
```