Programming Project #2: Image Quilting

```
## CS445: Computational Photography - Spring2020
```

In [1]:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook
import utils
import os
```

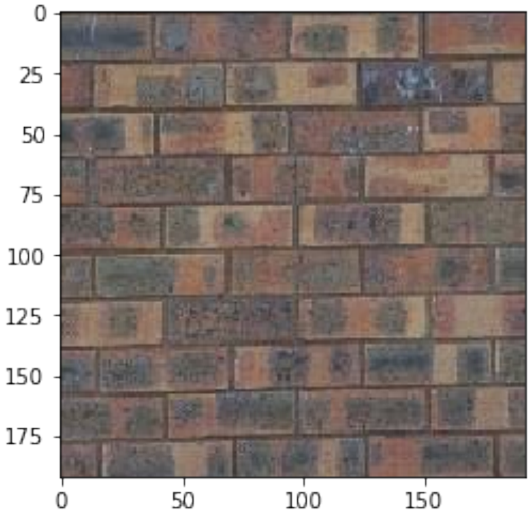
In [2]:

```
from utils import cut # default cut function for seam finding sec
```

Part I: Randomly Sampled Texture (10 pts)

In [13]:

```
sample_img_dir = 'samples/bricks_small.jpg' # feel free to change
sample_img = None
if os.path.exists(sample_img_dir):
    sample_img = cv2.imread(sample_img_dir)
    sample_img = cv2.cvtColor(sample_img, cv2.COLOR_BGR2RGB)
    plt.imshow(sample_img)
```

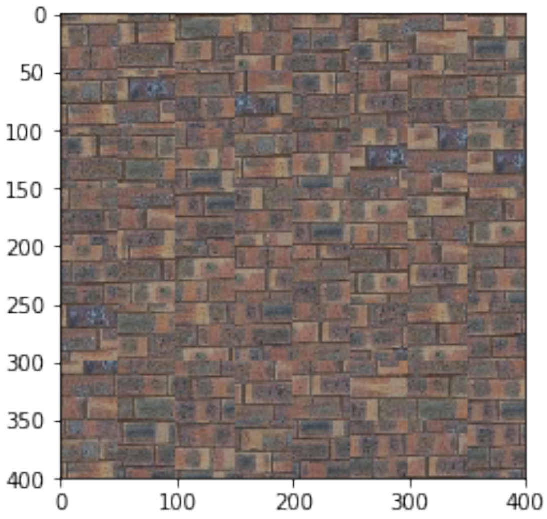


In [14]:

```
def quilt_random(sample, out_size, patch_size):
    """
    Randomly samples square patches of size patchsize from sample

    :param sample: numpy.ndarray   The image you read from sample
    :param out_size: int            The width of the square output
    :param patch_size: int          The width of the square sample
    :return: numpy.ndarray
    """

    # To do

    img_w = sample.shape[0]
    img_h = sample.shape[1]

    res = np.empty([out_size, out_size, 3], dtype=int)
    for i in range(0, out_size, patch_size):
        for j in range(0, out_size, patch_size):
            patch_w = random.randrange(img_w-patch_size)
            patch_h = random.randrange(img_h-patch_size)
            res[i:i+min(i+patch_size, out_size) - i, j:j+min(j+pat

    return res
```

In [15]:

```
import random
out_size = 400  # feel free to change to debug
patch_size = 50 # feel free to change to debug
res = quilt_random(sample_img, out_size, patch_size)
if res.any():
    plt.imshow(res)
```



Part II: Overlapping Patches (30 pts)

In [16]:

```
def ssd_patch(patchRes, mask, sample):
    sample = sample/255.0
    img_resolution = (sample.shape[0],sample.shape[1])
    img_channels = sample.shape[2]
    ssds = np.zeros(img_resolution)
    for i in range(img_channels):
        ssds += ((mask*patchRes[:,:,i])**2).sum() - 2 * cv2.filte
                                                          ddepth=

    return ssds/img_channels
```

In [17]:

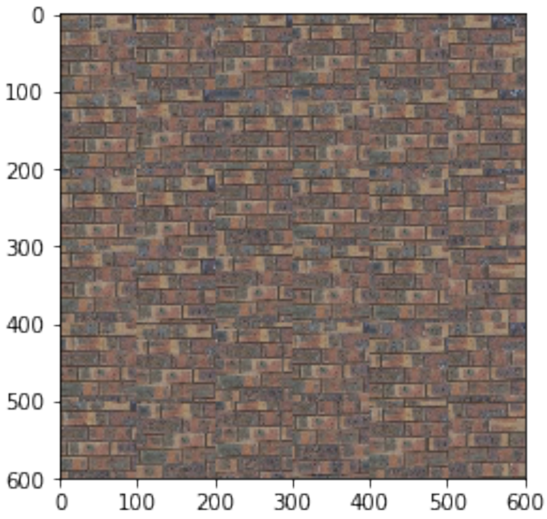
```
def choose_sample(ssd_template_matched, tol, r, c, K_lowest_cost_p
    k_low_cost_patches_res = []
    ssd_template_matched[:int(r/2)+1,:] = float("inf")
    ssd_template_matched[-(int(r/2)+1):,:] = float("inf")
    ssd_template_matched[:,:int(c/2)+1] = float("inf")
    ssd_template_matched[:,-(int(c/2)+1):] = float("inf")

    min_cost = np.amin(ssd_template_matched)
    row,col = np.where(ssd_template_matched <= min_cost*(1+tol))

    if (row.shape[0] > K_lowest_cost_patches):
        i = random.randrange(0,row.shape[0])
        return (row[i], col[i])


    while (len(k_low_cost_patches_res) < K_lowest_cost_patches):
        min_cost_idx = np.where(ssd_template_matched == np.amin(ss
        k_low_cost_patches_res.append([ min_cost_idx[0][0], min_co
        ssd_template_matched[min_cost_idx] = float("inf")

    return k_low_cost_patches_res[random.randrange(0,len(k_low_cos
```

In [18]:

```
def quilt_simple(sample, out_size, patch_size, overlap, tol):
    """
    Randomly samples square patches of size patchsize from sample
    Feel free to add function parameters
    :param sample: numpy.ndarray
    :param out_size: int
    :param patch_size: int
    :param overlap: int
    :param tol: float
    :return: numpy.ndarray
    """
    # Todo
    K_lowest_cost_patches = 5
    res = np.empty([out_size, out_size, 3], dtype=int)
    for i in range(0, out_size, patch_size-overlap):
        for j in range(0, out_size, patch_size-overlap):
            mask = np.zeros(((min(i+patch_size, out_size) - i), (
            if (i > 0):
                mask[0:overlap,:] = 1.0
            if (j > 0):
                mask[:,0:overlap] = 1.0
            ssd_template_matched = ssd_patch(res[i:i+(min(i+patch_
            row,column = choose_sample(ssd_template_matched, tol,
            res[i:i+(min(i+patch_size, out_size) - i), j:j+(min(j

    return res
```

In [19]:

```
res = quilt_simple(sample_img, 600, 150, 50, .0002) #feel free to
if res.any():
    plt.imshow(res)
```



Part III: Seam Finding (20 pts)

In [20]:

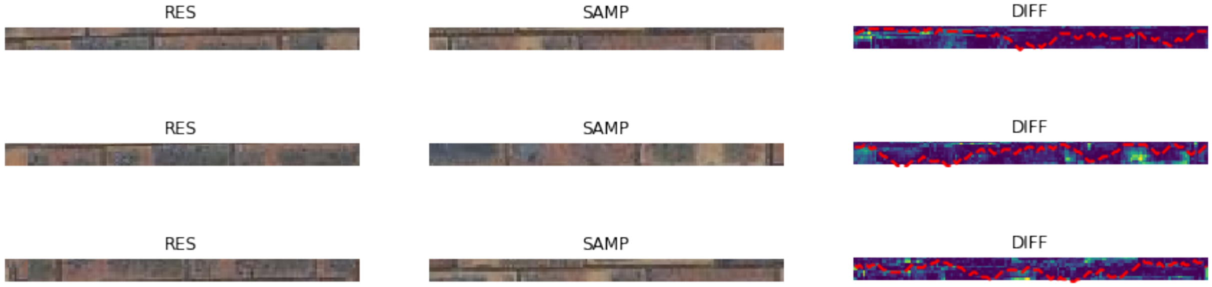
```
# optional or use cut(err_patch) directly
def customized_cut(bndcost):
    pass
```

In [21]:

```
def quilt_cut(sample, out_size, patch_size, overlap, tol):
    """
    Samples square patches of size patchsize from sample using seam
    Feel free to add function parameters
    :param sample: numpy.ndarray
    :param out_size: int
    :param patch_size: int
    :param overlap: int
    :param tol: float
    :return: numpy.ndarray
    """
    K_lowest_cost_patches = 2
    img_channels = sample.shape[2]

    res = np.empty([out_size, out_size, 3], dtype=int)
    for i in range(0, out_size-overlap, patch_size-overlap):
        for j in range(0, out_size-overlap, patch_size-overlap):

            mask = np.zeros(((min(i+patch_size, out_size) - i), (mir
            cut_mask = np.ones((patch_size, patch_size))

            if (i > 0):
                mask[0:overlap,:] = 1.0
            if (j > 0):
                mask[:,0:overlap] = 1.0

            ssd_template_matched = ssd_patch(res[i:i+(min(i+patch_si
                                             mask, sample/255.0)

            row,column = choose_sample(ssd_template_matched, tol,(mi
            if (i > 0):
                sdif = (res[i:i+overlap,j:j+(min(j+patch_size, out_s
                cut_mask[:overlap,:(min(j+patch_size, out_size) - j)


                resultShow = res[i:i+overlap,j:j+(min(j+patch_size,
                sampleShow = sample[row-int((min(i+patch_size, out_s
                differenceShow = sdif.sum(axis=2)



                fig = plt.figure(figsize = (15,5) )

                plt.subplot(1, 3, 1)
                plt.imshow(resultShow)
                plt.title("RES", fontsize=12)
                plt.axis('off')

                plt.subplot(1, 3, 2)
                plt.imshow(sampleShow)
                plt.title("SAMP", fontsize=12)
                plt.axis('off')

                plt.subplot(1, 3, 3)
                plt.imshow(differenceShow)
                plt.title("DIFF", fontsize=12)
                plt.axis('off')
                plt.plot(range((min(j+patch_size, out_size) - j)), b

                plt.show()






            if (j > 0):
                sdif = (res[i:i+(min(i+patch_size, out_size) - i),j:
                cut_mask[:(min(i+patch_size, out_size) - i),:overlap
            for c in range(img_channels):
                cut_img = sample[row-int((min(i+patch_size, out_size
                                 column-int((min(j+patch_size, out_s
                for k in range((min(i+patch_size, out_size) - i)):
                    for l in range((min(j+patch_size, out_size) - j)
                        if (cut_img[k,l] > 0):
                            res[i+k,j+l,c] = cut_img[k,l]

    return res
```

In [22]:

```
res = quilt_cut(sample_img, 600, 150, 10, 0.0002)
if res.any():
    plt.imshow(res)
```



RES
SAMP
DIFF

RES
SAMP
DIFF

RES
SAMP
DIFF

RES
SAMP
DIFF

RES
SAMP
DIFF

RES
SAMP
DIFF

RES
SAMP
DIFF

RES
SAMP
DIFF

RES
SAMP
DIFF

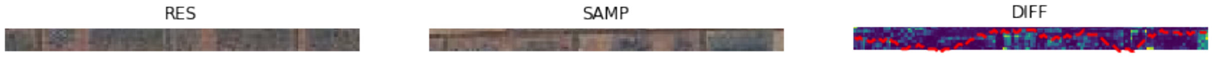
RES
SAMP
DIFF

RES
SAMP
DIFF

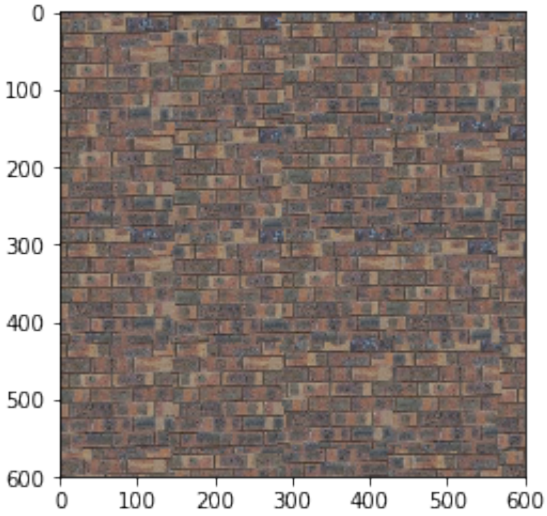
RES
SAMP
DIFF

RES
SAMP
DIFF

RES
SAMP
DIFF



RES
SAMP
DIFF



part IV: Texture Transfer (30 pts)

In [23]:

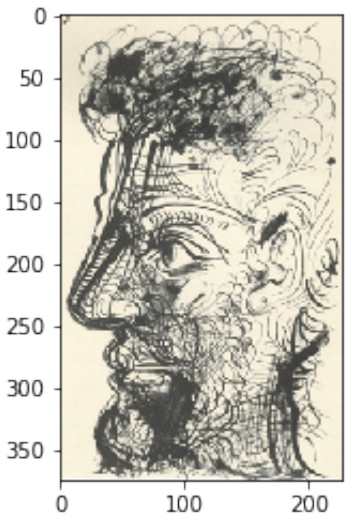
```
def ssd_patch_targeted(res, mask, sampleImg, targetImg, alpha):
    imgRes = ((sampleImg.shape[0], sampleImg.shape[1]))
    imgChannels = sampleImg.shape[2]
    ssds = np.zeros(imgRes)
    if (len(targetImg.shape) > 2):
        sampleImg = sampleImg/255.0
        for i in range(imgChannels):
            ssds += (alpha)*(((mask*res[:,:,i])**2).sum() - 2 * cv


            ssds += (1-alpha)*(((targetImg[:,:,i])**2).sum() - 2 *

        return ssds/imgChannels

    sampleImg = cv2.cvtColor(sampleImg, cv2.COLOR_BGR2GRAY) / 255.
    res = cv2.cvtColor(res, cv2.COLOR_BGR2GRAY)
    ssds += (alpha)*(((mask*res)**2).sum() - 2 * cv2.filter2D(samp
    ssds += (1-alpha)*(((targetImg)**2).sum() - 2 * cv2.filter2D(s
    return ssds/imgChannels
```

In [24]:

```
sampleImg = cv2.imread('samples/sketch.tiff')
sampleImg = cv2.cvtColor(sampleImg, cv2.COLOR_BGR2RGB)
plt.imshow(sampleImg)

```

Out[24]:

```
<matplotlib.image.AxesImage at 0x1fd61e5db70>
```

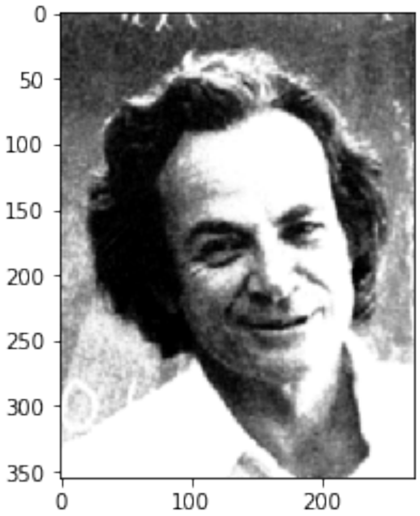


In [25]:

```
targetImg = cv2.imread('samples/feynman.tiff')
targetImg = cv2.cvtColor(targetImg, cv2.COLOR_BGR2RGB)
plt.imshow(targetImg)
```

Out[25]:

```
<matplotlib.image.AxesImage at 0x1fd619ff2e8>
```

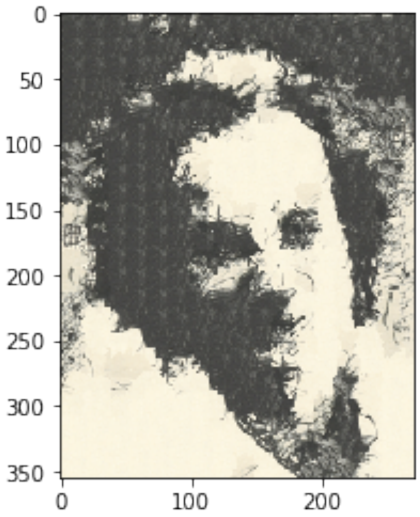


In [26]:

```python
def texture_transfer(sampleImg, targetImg, patch_size, overlap, to
    """
    Feel free to add function parameters
    """
    K_lowest_cost_patches = 2


    imgw, imgh, imgchannel = (targetImg.shape[0], targetImg.shape
    targetImg = cv2.cvtColor(targetImg, cv2.COLOR_RGB2GRAY)

    res = np.zeros((imgw,imgh, imgchannel), np.uint8)
    for i in range(0, imgw-overlap, patch_size-overlap):
        for j in range(0, imgh-overlap, patch_size-overlap):

            mask = np.zeros(((min(i+patch_size, imgw) - i), (min(j
            if (i > 0):
                mask[0:overlap,:] = 1.0
            if (j > 0):
                mask[:,0:overlap] = 1.0

            ssd_template_matched = ssd_patch_targeted(res[i:i+(min
                                                       targetImg[i:

            row,column = choose_sample(ssd_template_matched, tol,

            cut_mask = np.ones((patch_size, patch_size))

            if (i > 0):
                sdif = (res[i:i+overlap,j:j+(min(j+patch_size, img
                cut_mask[:overlap,:(min(j+patch_size, imgh) - j)]

            if (j > 0):
                sdif = (res[i:i+(min(i+patch_size, imgw) - i),j:j+
                cut_mask[:(min(i+patch_size, imgw) - i),:overlap]

            for c in range(imgchannel):
                cut_img = sampleImg[row-int((min(i+patch_size, img
                                    column-int((min(j+patch_size,
                for k in range((min(i+patch_size, imgw) - i)):
                    for l in range((min(j+patch_size, imgh) - j)):
                        if (cut_img[k,l] > 0):
                            res[i+k,j+l,c] = cut_img[k,l]

    return res
```

In [27]:

```
res = texture_transfer(sampleImg, targetImg, 24, 8, 0.00002, .0002
if res.any():
    plt.imshow(res)
```



Bells & Whistles

(10 pts) Create and use your own version of cut.m. To get these points, you should create your own implementation without basing it directly on the provided function (you're on the honor code for this one).

You can simply copy your customized_cut(bndcost) into the box below so that it is easier for us to grade

In [ ]:

```

```

(15 pts) Implement the iterative texture transfer method described in the paper. Compare to the non-iterative method for two examples.

In [28]:

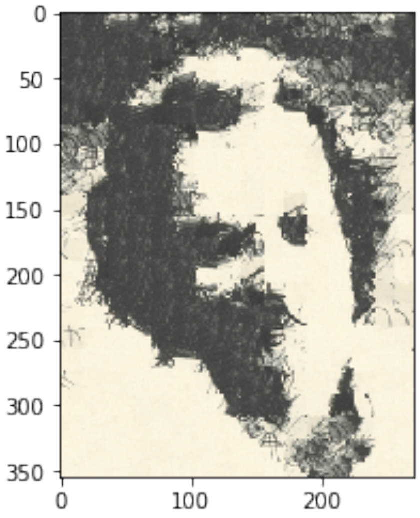
```
def iter_texture_transfer(sample, target, res_prev, patch_size, ov
    img_channels = sample.shape[2]
    res = res_prev.copy()
    for i in range(0, target.shape[0]-overlap, patch_size-overlap)
        for j in range(0, target.shape[1]-overlap, patch_size-over
            mask = np.ones(((min(i+patch_size, target.shape[0]) -
            if (np.count_nonzero(res) == 0):
                mask = np.zeros(((min(i+patch_size, target.shape[0
                if (i > 0):
                    mask[0:overlap,:] = 1.0
                if (j > 0):
                    mask[:,0:overlap] = 1.0

            ssd_template_matched = ssd_patch_targeted(res[i:i+(mi

            p,q = choose_sample(ssd_template_matched , tol, (min(i

            cut_mask = np.ones((patch_size, patch_size))

            if (i > 0):
                sdif = (res[i:i+overlap,j:j+(min(j+patch_size, ta
                cut_mask[:overlap,:(min(j+patch_size, target.shape

            if (j > 0):
                sdif = (res[i:i+(min(i+patch_size, target.shape[0]
                cut_mask[:(min(i+patch_size, target.shape[0]) - i)

            for c in range(img_channels):
                cut_img = sample[p-int((min(i+patch_size, target.s
                for k in range((min(i+patch_size, target.shape[0])
                    for l in range((min(j+patch_size, target.shape
                        if (cut_img[k,l] > 0):
                            res[i+k,j+l,c] = cut_img[k,l]

    return res
```

In [29]:

```
block_size = 128

targetImg_blured = cv2.cvtColor(cv2.blur(targetImg, (1,1)), cv2.C
resImg = np.zeros((targetImg_blured.shape[0], targetImg_blured.sha

for i in range(4):
    resImg = iter_texture_transfer(sampleImg, targetImg_blured, re
                                                 overlap = int(block_size/(2**i)
                                                    tol = 0.0002, K_lowest_cost_
plt.imshow(resImg)
```

Out[29]:

```
<matplotlib.image.AxesImage at 0x1fd61468320>
```



In [ ]:

```
```

In [ ]:

```
```

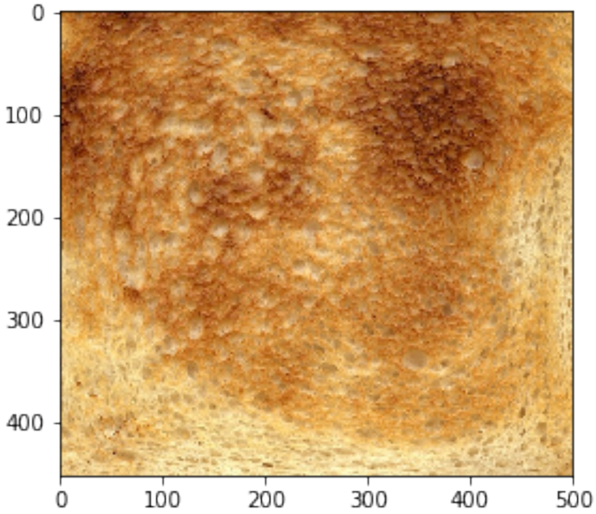
(up to 20 pts) Use a combination of texture transfer and blending to create a face-in-toast image like the one on top. To get full points, you must use some type of blending, such as feathering or Laplacian pyramid blending.

In [30]:

```
sampleImg = cv2.imread('samples/toast-sample.jpg')
sampleImg = cv2.cvtColor(sampleImg, cv2.COLOR_BGR2RGB)

plt.imshow(sampleImg)

```

Out[30]:

```
<matplotlib.image.AxesImage at 0x1fd60b168d0>
```

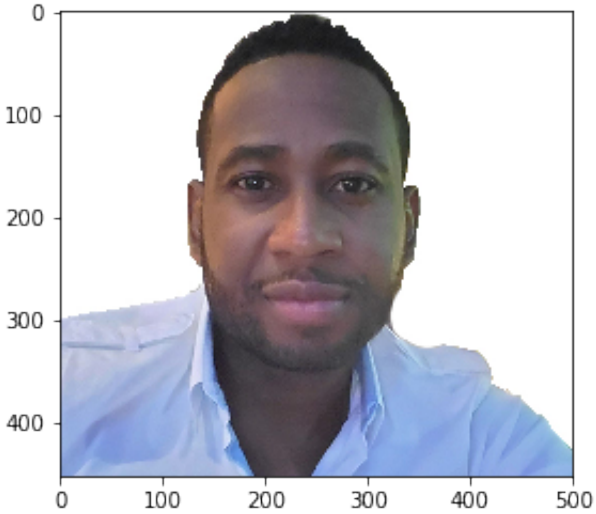


In [37]:

```
targetImg = cv2.imread('samples/my_photo.jpg')
targetImg = cv2.cvtColor(targetImg, cv2.COLOR_BGR2RGB)
plt.imshow(targetImg)
```

Out[37]:

```
<matplotlib.image.AxesImage at 0x1fd60ee8da0>
```

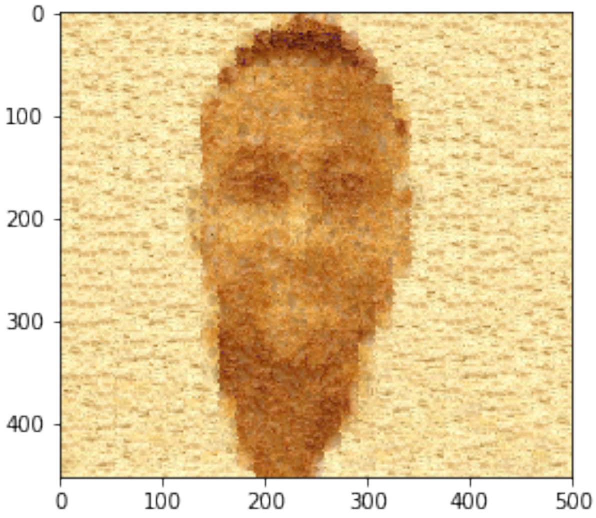


In [38]:

```
block_size = 128

targetImg_blured = cv2.cvtColor(cv2.blur(targetImg, (5,5)), cv2.CO
resImg = np.zeros((targetImg_blured.shape[0], targetImg_blured.sha

for i in range(4):
    resImg = iter_texture_transfer(sampleImg, targetImg_blured, re
                                   overlap = int(block_size/(2**i)
                                     tol = 0.0002, K_lowest_cost_p
plt.imshow(resImg)
```

Out[38]:

```
<matplotlib.image.AxesImage at 0x1fd605704a8>
```



In [39]:

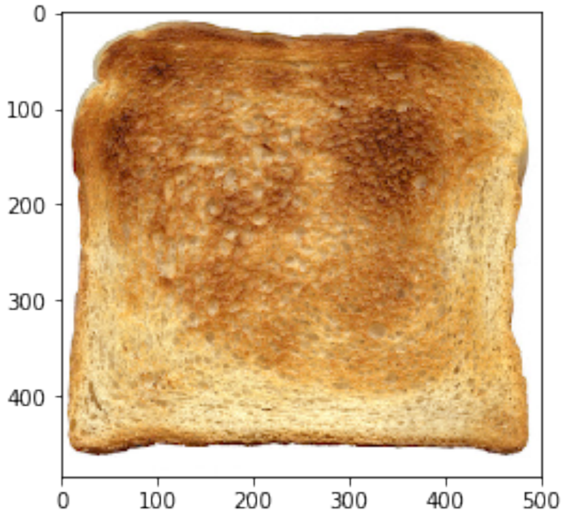
```
def blending_laplacian_pyramid(texture_result, toastImg, mask, lev

    texture_result_pyramid = [texture_result]
    toastImg_pyramid = [toastImg]

    mask_pyramid = [mask]

    for i in range(levels):
        texture_result = cv2.pyrDown(texture_result)
        toastImg = cv2.pyrDown(toastImg)
        mask = cv2.pyrDown(mask)
        texture_result_pyramid.append(np.float32(texture_result))
        toastImg_pyramid.append(np.float32(toastImg))
        mask_pyramid.append(np.float32(mask))

    laplacian_pyramid_texture  = [texture_result_pyramid[levels-1]
    laplacian_pyramid_toast  = [toastImg_pyramid[levels-1]]

    for i in range(levels-1, 0, -1):
        laplacian_pyramid_texture.append(np.subtract(texture_resul
        laplacian_pyramid_toast.append(np.subtract(toastImg_pyrami

    mask_pyramid.reverse()

    blending = []
    for texture,toast,msk in zip(laplacian_pyramid_texture,laplaci
        blending.append(texture * msk + toast * (1.0 - msk))

    out = blending[0]
    for i in range(1,levels):
        out = cv2.pyrUp(out)
        out = cv2.add(out[:blending[i].shape[0],:blending[i].shape

    return out
```

In [40]:

```
maskImg = cv2.imread('samples/my_mask.jpg')
maskImg = cv2.cvtColor(maskImg, cv2.COLOR_BGR2RGB)

toastImg = cv2.imread('samples/toast.jpg')
toastImg = cv2.cvtColor(toastImg, cv2.COLOR_BGR2RGB)
plt.imshow(toastImg)
```

Out[40]:

```
<matplotlib.image.AxesImage at 0x1fd6056f8d0>
```

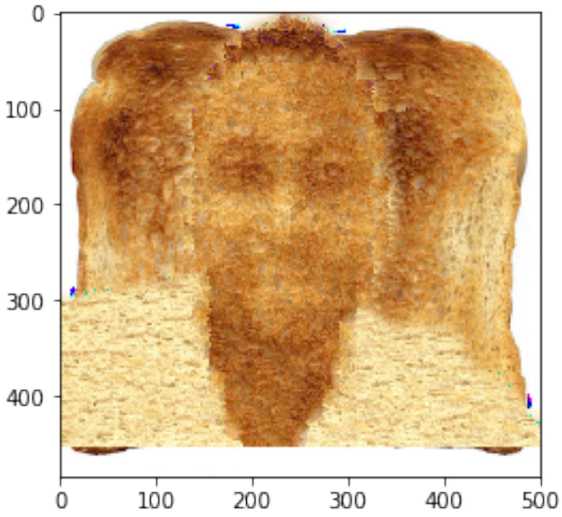


In [41]:

```
texture_result = resImg.copy()

imgw, imgh = (texture_result.shape[0], texture_result.shape[1])

final_output = toastImg.copy()
final_output[0:imgw,0:imgh] = blending_laplacian_pyramid(texture_

plt.imshow(final_output)
```

Out[41]:

```
<matplotlib.image.AxesImage at 0x1fd63083c18>
```



(up to 40 pts) Extend your method to fill holes of arbitrary shape for image completion. In this case, patches are drawn from other parts of the target image. For the full 40 pts, you should implement a smart priority function (e.g., similar to Criminisi et al.).