Programming Project #2: Image Quilting

CS445: Computational Photography

Texture synthesis is the creation of a larger texture image from a small sample. For texture synthesis, the main idea is to **sample patches** and lay them down in **overlapping patterns**, such that the overlapping regions are similar. The overlapping regions may not match exactly, which will result in noticeable edges. To fix this, you will compute a path along pixels with similar intensities through the overlapping region and use it to select which overlapping patch from which to draw each pixel.

Texture transfer is giving an object the appearance of having the same texture as a sample while preserving its basic shape. Texture transfer is achieved by encouraging **sampled patches** to have similar appearance to a given target image, as well as **matching overlapping regions** of already sampled patches. In this project, you will apply important techniques such as template matching, finding seams, and masking. These techniques are also useful for image stitching, image completion, image retargeting, and blending.

In [1]:
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import os
from random import random
import time
import utils
from utils import cut # default cut function for seam finding section
```

Part I: Randomly Sampled Texture (10 pts)

In [4]:
```
def quilt_random(sample, out_size, patch_size):
    """
    Randomly samples square patches of size patchsize from sample in order to create an

    :param sample: numpy.ndarray   The image you read from sample directory
    :param out_size: int            The width of the square output image
    :param patch_size: int          The width of the square sample patch
    :return: numpy.ndarray
    """
    out = np.zeros((out_size, out_size, 3), np.uint8)
    patches = out_size//patch_size
    sh, sw, _ = sample.shape

    for i in range(patches):
        for j in range(patches):
            x = np.random.randint(0, sw-patch_size)
            y = np.random.randint(0, sh-patch_size)
            out[i*patch_size:(i+1)*patch_size, j*patch_size:(j+1)*patch_size, :] = sampl

    return out
```

In [5]:
```
out_size = 200  # change these parameters as needed
patch_size = 20

# load sample img
```

```
sample_img_fn = 'samples/bricks_small.jpg' # feel free to change
sample_img = cv2.cvtColor(cv2.imread(sample_img_fn), cv2.COLOR_BGR2RGB)
print("sample img size:", sample_img.shape)

# quilt random
res = quilt_random(sample_img, out_size, patch_size)
# res = cv2.cvtColor(res.astype(np.uint8), cv2.COLOR_BGR2RGB)
print("quilt image size", res.shape)

# plot
fig, axes = plt.subplots(1,2, figsize=(10,5))
axes[0].imshow(sample_img)
axes[0].set_title("sample image")
axes[1].imshow(res)
axes[1].set_title("quilt random image")
```

```
sample img size: (192, 192, 3)
quilt image size (200, 200, 3)
```

Out[5]:

```
Text(0.5, 1.0, 'quilt random image')
```

Part II: Overlapping Patches (30 pts)

In [7]:

```
def ssd_patch(T, M, I, edges_cost = True):
    """
    Performs template matching with the overlapping region, computing the cost of sampli

    :param T: numpy.ndarray   The template, which is the patch in the current output ima
    :param M: numpy.ndarray   The mask, which has values 1 in the overlapping region, an
    :param I: numpy.ndarray   The input texture image.
    return cost_map: numpy.ndarray   The cost of sampling each patch.
    """
    T, M, I = T.astype(np.float64), M.astype(np.float64), I.astype(np.float64)
    # Compute the SSD for each channel.
    # Each pixel of the ssd_cost gives you the cost for sampling a patch centered around
    ssd_costs = []
    for i in range(3):
        cost = ((M[:,:,i] * (T[:,:,i]))**2).sum() - 2 * cv2.filter2D(I[:,:,i], ddepth=-1
        ssd_costs.append(cost)
    cost_map = np.array(ssd_costs).sum(axis=0)

    # Set the ssd cost at the edges as inf
    if edges_cost:
```

```
        patch_size = T.shape[0]
        cost_map[:patch_size//2,:] = np.inf # top
        cost_map[-patch_size//2:,:] = np.inf # bottom
        cost_map[:,:patch_size//2] = np.inf # left
        cost_map[:,-patch_size//2:] = np.inf # right

    return np.absolute(cost_map)
```

In [8]:
```
def choose_sample(cost_map, tol):
    """
    Randomly selects one of the top few samples to avoid getting stuck in local minima i

    :param cost_map: numpy.ndarray   The cost of sampling each patch.
    :param tol: int                  A tolerance parameter that specifies how many of th
    :return: tuple                   The coordinates of the top left corner of the patch
    """
    # Find the top few minimum cost samples.
    top_min = np.argpartition(cost_map, tol, axis=None)[:tol]
    # Randomly choose one of the top few minimum cost samples.
    np.random.seed(0)
    idx = np.random.randint(0, tol)
    # Convert the index into a coordinate.
    y, x = np.unravel_index(top_min[idx], cost_map.shape)
    return (y, x)
```

In [8]:
```
def quilt_simple(sample, out_size, patch_size, overlap, tol):
    """
    Randomly samples square patches of size patchsize from sample in order to create an
    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
    """
    out = np.zeros((out_size, out_size, 3), np.uint8)
    sh, sw, _ = sample.shape
    step = patch_size - overlap
    num_steps = out_size//step-1
    half = patch_size//2

    # start by randomly sampling a patch for the upper-left corner
    np.random.seed(0)
    x0 = np.random.randint(0, sw-patch_size)
    y0 = np.random.randint(0, sh-patch_size)
    patch0 = sample[y0:y0+patch_size, x0:x0+patch_size, :]
    out[:patch_size, :patch_size, :] = patch0

    # fill in the rest of the image in raster scan order
    for i in range(num_steps):
        for j in range(num_steps):
            # set mask
            if i == 0 and j == 0:
                continue
            elif i == 0: # first row
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
                mask[:, :overlap, :] = 1
            elif j == 0: # first column
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
                mask[:overlap, :, :] = 1
            else:
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
```

```
                mask[:, :overlap, :] = 1
                mask[:overlap, :, :] = 1

            # sample new patches and overlap with the existing ones
            template = out[i*step:i*step+patch_size, j*step:j*step+patch_size, :]
            cost_map = ssd_patch(template, mask, sample, True)
            patch_y, patch_x = choose_sample(cost_map, tol)
            if patch_size%2 == 0:
                patch = sample[patch_y-half:patch_y+half, patch_x-half:patch_x+half, :]
            else:
                patch = sample[patch_y-half:patch_y+half+1, patch_x-half:patch_x+half+1,
            out[i*step:i*step+patch_size, j*step:j*step+patch_size, :] = patch

    return out
```

In [9]:

```
sample_img_fn = 'samples/bricks_small.jpg'
sample_img = cv2.cvtColor(cv2.imread(sample_img_fn), cv2.COLOR_BGR2RGB)
plt.imshow(sample_img)
plt.show()

out_size = 300  # change these parameters as needed
patch_size = 23
overlap = 11
tol = 5
res = quilt_simple(sample_img, out_size, patch_size, overlap, tol) #feel free to change

plt.figure(figsize=(10,10))
plt.imshow(res)
```

Out[9]: <matplotlib.image.AxesImage at 0x128d530d0>

Part III: Seam Finding (20 pts)

In [10]:
```
# optional or use cut(err_patch) directly
def customized_cut(bndcost):
    pass
```

In [11]:
```
def quilt_cut(sample, out_size, patch_size, overlap, tol):
    """
    Samples square patches of size patchsize from sample using seam finding in order to
    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
    """
    out = np.zeros((out_size, out_size, 3), np.uint8)
    sh, sw, _ = sample.shape
    step = patch_size - overlap
```

```
    num_steps = out_size//step-1
    half = patch_size//2

    # start by randomly sampling a patch for the upper-left corner
    np.random.seed(0)
    x0 = np.random.randint(0, sw-patch_size)
    y0 = np.random.randint(0, sh-patch_size)
    patch0 = sample[y0:y0+patch_size, x0:x0+patch_size, :]
    out[:patch_size, :patch_size, :] = patch0

    # save for analysis
    # patches = []
    # templates = []
    # cut_masks = []

    # randomly choose one patch for illustration
    # not in the first row or column
    np.random.seed(0)
    a = np.random.randint(1, num_steps)
    b = np.random.randint(1, num_steps)
    out_dict = {}

    # fill in the rest of the image in raster scan order
    for i in range(num_steps):
        for j in range(num_steps):
            if i == 0 and j == 0:
                continue
            elif i == 0: # first row
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
                mask[:, :overlap, :] = 1
            elif j == 0: # first column
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
                mask[:overlap, :, :] = 1
            else:
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
                mask[:, :overlap, :] = 1
                mask[:overlap, :, :] = 1

            # sample new patches
            template = out[i*step:i*step+patch_size, j*step:j*step+patch_size, :]
            cost_map = ssd_patch(template, mask, sample, True)
            patch_y, patch_x = choose_sample(cost_map, tol)
            if patch_size%2 == 0:
                patch = sample[patch_y-half:patch_y+half, patch_x-half:patch_x+half, :]
            else:
                patch = sample[patch_y-half:patch_y+half+1, patch_x-half:patch_x+half+1,

            template_2 = template.copy()
            patch_2 = patch.copy()
            # find the min-cost contiguous path
            if i == 0: # first row
                err_patch = ssd_patch(template, mask, patch, False)[:, :overlap]
                cut_mask = cut(err_patch.T).T # vertical path
                for c in range(3):
                    patch_2[:, :overlap, c] = cut_mask * patch[:, :overlap, c]
                    template_2[:, :overlap, c] = (1 - cut_mask) * template[:, :overlap,

            elif j == 0: # first column
                err_patch = ssd_patch(template, mask, patch, False)[:overlap, :]
                cut_mask = cut(err_patch) # horizontal path
                for c in range(3):
                    patch_2[:overlap, :, c] = cut_mask * patch[:overlap, :, c]
                    template_2[:overlap, :, c] = (1 - cut_mask) * template[:overlap, :,

            else:
                err_patch_1 = ssd_patch(template, mask, patch, False)[:, :overlap]
```

```
                mask_1 = cut(err_patch_1.T).T
                err_patch_2 = ssd_patch(template, mask, patch, False)[:overlap, :]
                mask_2 = cut(err_patch_2)
                cut_mask = np.ones((patch_size, patch_size), np.uint8)
                cut_mask[:, :overlap] = mask_1 * cut_mask[:, :overlap]
                cut_mask[:overlap, :] = mask_2 * cut_mask[:overlap, :]

                # save for analysis
                if i == a and j == b:
                    out_dict['sample_patch'] = patch.copy()
                    out_dict['template'] = template.copy()
                    out_dict['err_patch_v'] = err_patch_1.copy()
                    out_dict['err_patch_h'] = err_patch_2.copy()
                    out_dict['mask_v'] = mask_1.copy()
                    out_dict['mask_h'] = mask_2.copy()
                    out_dict['mask'] = cut_mask.copy()

                for c in range(3):
                    patch_2[:, :, c] = cut_mask * patch[:, :, c]
                    template_2[:, :, c] = (1 - cut_mask) * template[:, :, c]

            # patches.append(patch_2)
            # templates.append(template_2)
            # cut_masks.append(cut_mask)

            new_patch = patch_2 + template_2
            out[i*step:i*step+patch_size, j*step:j*step+patch_size, :] = new_patch

    return out, out_dict
```

In [12]:

```
sample_img_fn = 'samples/bricks_small.jpg'
sample_img = cv2.cvtColor(cv2.imread(sample_img_fn), cv2.COLOR_BGR2RGB)
plt.imshow(sample_img)
plt.show()

out_size = 300  # change these parameters as needed
patch_size = 23
overlap = 11
tol = 5
res, out_dict = quilt_cut(sample_img, out_size, patch_size, overlap, tol)
# res = quilt_cut(sample_img, out_size, patch_size, overlap, tol)
if res is not None:
    plt.figure(figsize=(10,10))
    plt.imshow(res)
```

0
25
50
75
100
125
150
175
0
25
50
75
100
125
150
175

In [13]:

```
# For illustration (a) two over lapping portions
fig, axes = plt.subplots(1,2, figsize=(10,5))
axes[0].imshow(out_dict['template'])
axes[0].set_title("template")

mask = np.zeros((patch_size, patch_size, 3), np.float32)
mask[:, :overlap, :] = 1
mask[:overlap, :, :] = 1
axes[1].imshow((mask*out_dict['sample_patch']).astype(np.uint8))
axes[1].set_title("sample patch")
```

Out[13]:

```
Text(0.5, 1.0, 'sample patch')
```

In [14]:
```
# For illustration (b) pixel-wise SSD cost
fig, axes = plt.subplots(1,2, figsize=(10,5))
axes[0].imshow(out_dict['err_patch_v'])
axes[0].set_title("vertical SSD cost")
axes[1].imshow(out_dict['err_patch_h'])
axes[1].set_title("horizontal SSD cost")
```

Out[14]:
```
Text(0.5, 1.0, 'horizontal SSD cost')
```

In [15]:
```
# For illustration (c) horizontal mask; (d) vertical mask; (e) combination mask
fig, axes = plt.subplots(1,3, figsize=(15,5))
axes[0].imshow(out_dict['mask_v'])
axes[0].set_title("vertical mask")
axes[1].imshow(out_dict['mask_h'])
axes[1].set_title("horizontal mask")
```

```
axes[2].imshow(out_dict['mask'])
axes[2].set_title("combination mask")
```

Out[15]:

```
Text(0.5, 1.0, 'combination mask')
```

Two Additional Results

In [17]:

```
sample_img_fn = 'texture1.jpg'
sample_img = cv2.cvtColor(cv2.imread(sample_img_fn), cv2.COLOR_BGR2RGB)
sample_img = cv2.resize(sample_img, (200, 200))

out_size = 300  # change these parameters as needed
patch_size = 23
overlap = 11
tol = 3
res_random = quilt_random(sample_img, out_size, patch_size)
res_simple = quilt_simple(sample_img, out_size, patch_size, overlap, tol)
res_cut, out_dict = quilt_cut(sample_img, out_size, patch_size, overlap, tol)

fig, axes = plt.subplots(2,2, figsize=(15,15))
axes[0,0].imshow(sample_img)
axes[0,0].set_title("sample image")
axes[0,1].imshow(res_random)
axes[0,1].set_title("quilt random")
axes[1,0].imshow(res_simple)
axes[1,0].set_title("quilt simple")
axes[1,1].imshow(res_cut)
axes[1,1].set_title("quilt cut")
```

Out[17]:

```
Text(0.5, 1.0, 'quilt cut')
```

In [18]:
```
sample_img_fn = 'texture2.jpg'
sample_img = cv2.cvtColor(cv2.imread(sample_img_fn), cv2.COLOR_BGR2RGB)
sample_img = cv2.resize(sample_img, (200, 200))

out_size = 300  # change these parameters as needed
patch_size = 23
overlap = 11
tol = 3
res_random = quilt_random(sample_img, out_size, patch_size)
res_simple = quilt_simple(sample_img, out_size, patch_size, overlap, tol)
res_cut, out_dict = quilt_cut(sample_img, out_size, patch_size, overlap, tol)

fig, axes = plt.subplots(2,2, figsize=(15,15))
axes[0,0].imshow(sample_img)
axes[0,0].set_title("sample image")
axes[0,1].imshow(res_random)
axes[0,1].set_title("quilt random")
axes[1,0].imshow(res_simple)
axes[1,0].set_title("quilt simple")
axes[1,1].imshow(res_cut)
axes[1,1].set_title("quilt cut")
```

Out[18]: Text(0.5, 1.0, 'quilt cut')

part IV: Texture Transfer (30 pts)

Your final task is to create a function texture_transfer, based on your quilt_cut function for creating a texture sample that is guided by a pair of sample/target correspondence images (section 3 of the paper). You do not need to implement the iterative method described in the paper (you can do so for extra points: see Bells and Whistles). The main difference between this function and quilt_cut is that there is an additional cost term based on the difference between the sampled source patch and the target patch at the location to be filled.

In [9]:
```
def texture_transfer(sample, patch_size, overlap, tol, guidance_im, alpha):
    """
    Samples square patches of size patchsize from sample using seam finding in order to
    Feel free to modify function parameters
    :param sample: numpy.ndarray
    :param patch_size: int
```

```
    :param overlap: int
    :param tol: float
    :param guidance_im: target overall appearance for the output
    :param alpha: float 0-1 for strength of target
    :return: numpy.ndarray
    """
    out_height, out_width, _ = guidance_im.shape
    out = np.zeros((out_height, out_width, 3), np.float32)
    sh, sw, _ = sample.shape
    step = patch_size - overlap
    half = patch_size//2

    # fill in the rest of the image in raster scan order
    for i in range(out_height//step-1):
        for j in range(out_width//step-1):
            if i == 0 and j == 0:
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
            elif i == 0: # first row
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
                mask[:, :overlap, :] = 1
            elif j == 0: # first column
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
                mask[:overlap, :, :] = 1
            else:
                mask = np.zeros((patch_size, patch_size, 3), np.float32)
                mask[:, :overlap, :] = 1
                mask[:overlap, :, :] = 1

            # compute the cost map for template and corresponding target patch
            template = out[i*step:i*step+patch_size, j*step:j*step+patch_size, :]
            target_patch = guidance_im[i*step:i*step+patch_size, j*step:j*step+patch_siz

            ssd_overlap = ssd_patch(template, mask, sample, True)
            ssd_transfer = ssd_patch(target_patch, mask, sample, True)
            cost_map = alpha * ssd_overlap + (1-alpha) * ssd_transfer

            # sample new patches
            patch_y, patch_x = choose_sample(cost_map, tol)
            if patch_size%2 == 0:
                patch = sample[patch_y-half:patch_y+half, patch_x-half:patch_x+half, :]
            else:
                patch = sample[patch_y-half:patch_y+half+1, patch_x-half:patch_x+half+1,

            template_2 = template.copy()
            patch_2 = patch.copy()
            # find the min-cost contiguous path
            if i == 0: # first row
                err_patch = ssd_patch(template, mask, patch, False)[:, :overlap]
                cut_mask = cut(err_patch.T).T # vertical path
                for c in range(3):
                    patch_2[:, :overlap, c] = cut_mask * patch[:, :overlap, c]
                    template_2[:, :overlap, c] = (1 - cut_mask) * template[:, :overlap,

            elif j == 0: # first column
                err_patch = ssd_patch(template, mask, patch, False)[:overlap, :]
                cut_mask = cut(err_patch) # horizontal path
                for c in range(3):
                    patch_2[:overlap, :, c] = cut_mask * patch[:overlap, :, c]
                    template_2[:overlap, :, c] = (1 - cut_mask) * template[:overlap, :,

            else:
                err_patch_1 = ssd_patch(template, mask, patch, False)[:, :overlap]
                mask_1 = cut(err_patch_1.T).T
                err_patch_2 = ssd_patch(template, mask, patch, False)[:overlap, :]
                mask_2 = cut(err_patch_2)
                cut_mask = np.ones((patch_size, patch_size), np.uint8)
```

```
                cut_mask[:, :overlap] = mask_1 * cut_mask[:, :overlap]
                cut_mask[:overlap, :] = mask_2 * cut_mask[:overlap, :]
                for c in range(3):
                    patch_2[:, :, c] = cut_mask * patch[:, :, c]
                    template_2[:, :, c] = (1 - cut_mask) * template[:, :, c]

            new_patch = patch_2 + template_2
            out[i*step:i*step+patch_size, j*step:j*step+patch_size, :] = new_patch

    return out
```

In [73]:

```
# load/process appropriate input texture and guidance images
# texture_img_fn = 'samples/sketch.tiff'
texture_img_fn = 'samples/white_small.jpg'
texture_img = cv2.cvtColor(cv2.imread(texture_img_fn), cv2.COLOR_BGR2RGB)

guidance_img_fn = 'smile.jpg'
guidance_img = cv2.cvtColor(cv2.imread(guidance_img_fn), cv2.COLOR_BGR2RGB)
h, w, _ = guidance_img.shape
guidance_img = cv2.resize(guidance_img, (w//8, h//8))

# guidance_img_fn = 'samples/feynman.tiff'
# guidance_img = cv2.cvtColor(cv2.imread(guidance_img_fn), cv2.COLOR_BGR2RGB)
```

In [57]:

```
patch_size = 25
overlap = 11
tol = 3
alpha = 0.5
res = texture_transfer(texture_img, patch_size, overlap, tol, guidance_img, alpha)
res = res.astype(np.uint8)

fig, axes = plt.subplots(1,3, figsize=(15,5))
axes[0].imshow(texture_img)
axes[0].set_title("texture image")
axes[1].imshow(guidance_img)
axes[1].set_title("guidance image")
axes[2].imshow(res)
axes[2].set_title("texture transfer image")
```

Out[57]:

```
Text(0.5, 1.0, 'texture transfer image')
```

Bells & Whistles

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

In [104…

```python
def iterative_texture_transfer(sample, patch_size, overlap, tol, guidance_im, num_iter):
    """
    Samples square patches of size patchsize from sample using seam finding in order to

    :param sample: numpy.ndarray
    :param patch_size: int
    :param overlap: int
    :param tol: float
    :param guidance_im: target overall appearance for the output
    :param num_iter: int
    :return: numpy.ndarray
    """
    source = sample.copy()
    for iter in range(num_iter):
        alpha = 0.1 + 0.8 * (iter)/(num_iter-1)
        print(f"iter: {iter}, patch_size: {patch_size}, overlap: {overlap}, alpha: {alph
        res = texture_transfer(source, patch_size, overlap, tol, guidance_im, alpha)

        # remove the black edges from the result
        gray = cv2.cvtColor(res.astype(np.uint8), cv2.COLOR_RGB2GRAY)
        _, thresh = cv2.threshold(gray, 1, 255, cv2.THRESH_BINARY)
        contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
        cnt = contours[0]
        x,y,w,h = cv2.boundingRect(cnt)
        res = res[y:y+h, x:x+w]

        # update the parameters
        patch_size = patch_size//3
        overlap = overlap//3
        source = res.copy()
    return res
```

In [108…

```python
num_iter = 3
patch_size = 50
overlap = 22
tol = 3
res = iterative_texture_transfer(texture_img, patch_size, overlap, tol, guidance_img, nu
res = res.astype(np.uint8)

fig, axes = plt.subplots(1,3, figsize=(15,5))
axes[0].imshow(texture_img)
axes[0].set_title("texture image")
axes[1].imshow(guidance_img)
axes[1].set_title("guidance image")
axes[2].imshow(res)
axes[2].set_title("iterative texture transfer image")
```

```
iter: 0, patch_size: 50, overlap: 22, alpha: 0.1
iter: 1, patch_size: 16, overlap: 7, alpha: 0.5
iter: 2, patch_size: 5, overlap: 2, alpha: 0.9
```

Out[108]:

```
Text(0.5, 1.0, 'iterative texture transfer image')
```

(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 [10]:
```
# load/process appropriate input texture and guidance images
texture_img_fn = 'samples/toast.jpg'
texture_img = cv2.cvtColor(cv2.imread(texture_img_fn), cv2.COLOR_BGR2RGB)

guidance_img_fn = 'smile.jpg'
guidance_img = cv2.cvtColor(cv2.imread(guidance_img_fn), cv2.COLOR_BGR2RGB)
h, w, _ = guidance_img.shape
guidance_img = cv2.resize(guidance_img, (w//8, h//8))
```

In [11]:
```
# Apply texture transfer to create the image with toast texture
patch_size = 25
overlap = 11
tol = 3
alpha = 0.5
res = texture_transfer(texture_img, patch_size, overlap, tol, guidance_img, alpha)
plt.imshow(res.astype(np.uint8))
```

Out[11]:
```
<matplotlib.image.AxesImage at 0x14bd35b20>
```

In [12]:
```
# crop the face region to square
face = res.astype(np.uint8).copy()
face = face[0:300, 30:330, :]
plt.imshow(face)
```

Out[12]:
```
<matplotlib.image.AxesImage at 0x14f15d100>
```

In [70]:
```
# blend the face with the toast texture
toast = texture_img.copy()
cy, cx, _ = toast.shape
fy, fx, _ = face.shape
```

```
canvas = np.zeros((cy, cx, 3))
canvas[cy//2-fy//2:cy//2+fy//2, cx//2-fx//2:cx//2+fx//2, :] = face

mask = np.zeros((cy, cx, 3))
num_mask_edge = 30
mask[cy//2-fy//2+num_mask_edge:cy//2+fy//2-num_mask_edge, cx//2-fx//2+num_mask_edge:cx//
blurred_mask = cv2.GaussianBlur(mask, (25, 25), 0)
# plt.imshow(blurred_mask)
# plt.show()

output = canvas * blurred_mask + toast * (1 - blurred_mask)

plt.imshow(output.astype(np.uint8))
```

Out[70]:

```
<matplotlib.image.AxesImage at 0x162e663a0>
```

In [72]:

```
fig, axes = plt.subplots(1,3, figsize=(15,5))
axes[0].imshow(texture_img)
axes[0].set_title("toast image")
axes[1].imshow(guidance_img)
axes[1].set_title("face image")
axes[2].imshow(output.astype(np.uint8))
axes[2].set_title("final image")
```

Out[72]:

```
Text(0.5, 1.0, 'final image')
```

(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.).

In [ ]:

In [ ]:

In [ ]:

In [ ]: