Homework 6

Fangda Li
ECE 661 - Computer Vision

October 25, 2016

1 Methodology

1.1 Otsu's Algorithm

In this subsection, the steps in my implementation of Otsu's algorithm for foreground segmentation in RGB images are described:

1. Split the original color image into three gray scale images, each representing one color channel. For each gray scale image, the following steps are applied to obtain a foreground mask, where background pixels are masked off. The final foreground mask is then obtained from logical operations of the three masks.

2. Obtain the probability distribution of pixel values in the gray scale image using histogram, as shown in Equation 1,

$$p_i = \frac{n_i}{N}, p_i \geq 0, \sum_{i=0}^{L} p_i = 1, \tag{1}$$

   where p_i is the pdf of pixel value i, n_i is the number of pixels with value i, N is the total number of pixels in the image, and L is the allowed maximum gray level (255 in this experiment).

3. Define the zeroth- and first-order cumulative moments of the probability distribution up to gray level k:

$$\omega(k) = \sum_{i=0}^{k} p_i, \tag{2}$$

$$\mu(k) = \sum_{i=0}^{k} i p_i. \tag{3}$$

   Calculate $\omega(k)$ and $\mu(k)$ with current k and the total mean gray level of the current image:

$$\mu_T = \sum_{i=0}^{L} i p_i. \tag{4}$$

4. Iterate through all the possible k values ($k \in [0, 255]$) to find their corresponding between-class variance, σ_B^2, where σ_B^2 is defined as:

$$\sigma_B^2(k) = \frac{[\mu_T \omega(k) - \mu(k)]^2}{\omega(k)[1 - \omega(k)]}. \tag{5}$$

Assign the foreground threshold k^* to be the k that maximizes σ_B^2:

$$k^* = \arg\max_k \sigma_B^2(k) \tag{6}$$

Obtain the foreground mask such that only the pixels with value greater than k^* are passed.

5. Repeat step (2)-(4) for multiple iterations to get more refined foreground masks if necessary. Combine the foreground masks from the three channels by logical operations to obtain the final output foreground mask.

1.2 Texture-based Feature Extraction and Segmentation

Often times, the pixel values of the foreground object are similar to those of the background pixels, which proposes a challenge for mere pixel value based foreground extraction methods. As a result, we exploit the spatial information of pixels using texture-based feature and represent the original image in texture space. More specifically, the texture-based feature for a pixel in this experiment is defined as the variance of the pixels values in a $N \times N$ neighboring window. In addition, we evaluate the original image with three different window sizes N to obtain three channels for the texture space image representation, similar to the RGB channels in a normal color image.

For foreground extraction, we simply apply Otsu's algorithm described in the previous subsection to the 3-channel texture space image.

1.3 Contour Extraction

Since we are just working with binary images, a contour pixel is simply a 255-valued pixel with more than one 0-valued pixels in its 3×3 neighborhood.

2 Discussion and Results

2.1 Discussion

1. There is no absolute "better" between color-based and texture-based approaches. For the leopard image, texture-based segmentation works better than color-based, since the orange pixels on the leopard are very similar to those of the ground. However, the opposite applies for the lake and brain images. Especially for the brain image, the textures of the white and gray matter are not distinguishable at all.

2. Obtaining the texture space image consumes significant amount of time due to its raster scanning operation across the whole image three times.

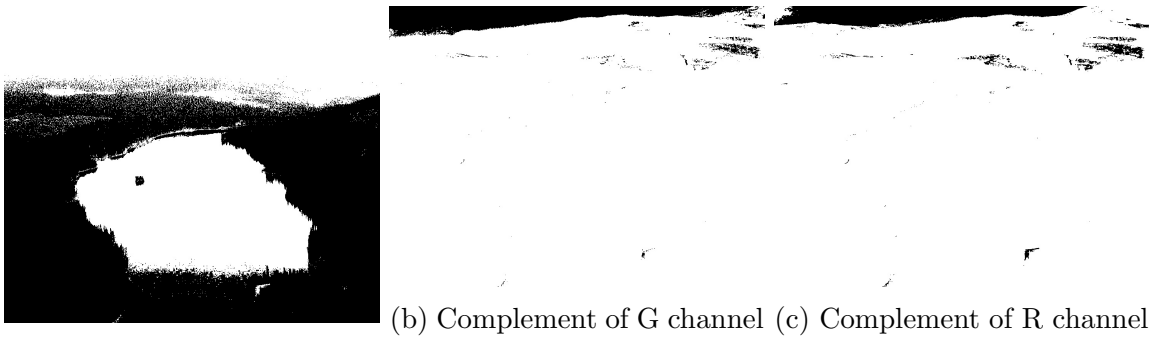
3. Applying morphological operations on the segmented foreground from Otsu's algorithm can help remove background noises by erosion then dilation. Then, applying dilation then erosion helps fill in the holes in the foreground, especially in the leopard image

4. For the lake image, the foreground is the most distinguishable in B channel. As a result, we *AND* the B channel mask with both the complements of G and R channel masks.

5. For the brain image, multiple iterations of Otsu's algorithm are necessary to separate the white matter from the gray matter.

2.2 Lake

Figure 1: Original image



(a) B channel mask

(b) Complement of G channel mask

(c) Complement of R channel mask

Figure 2: Masks; the final mask is obtained by ANDing all three masks.

Figure 3: Segmented foreground

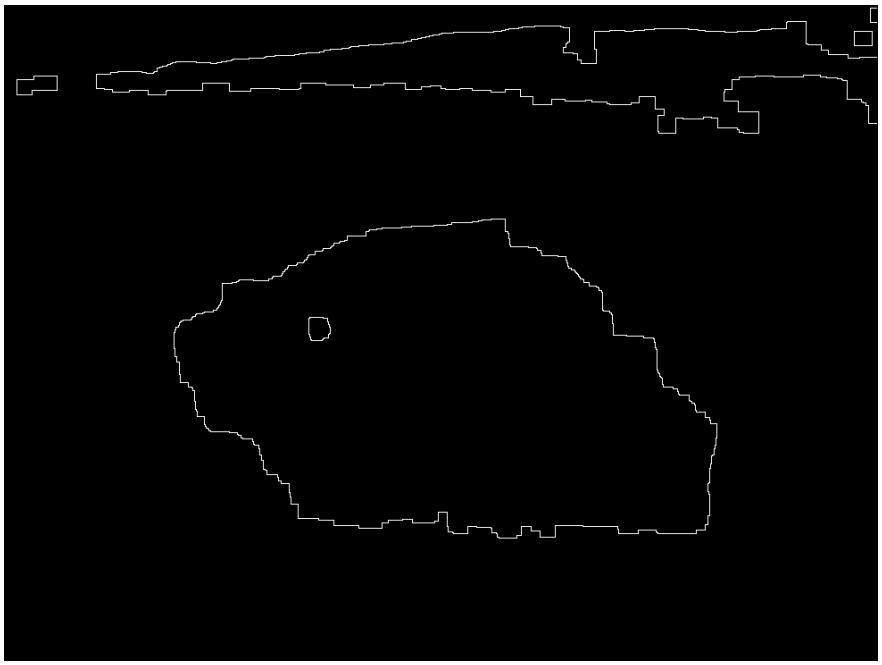


Figure 4: Contour extraction

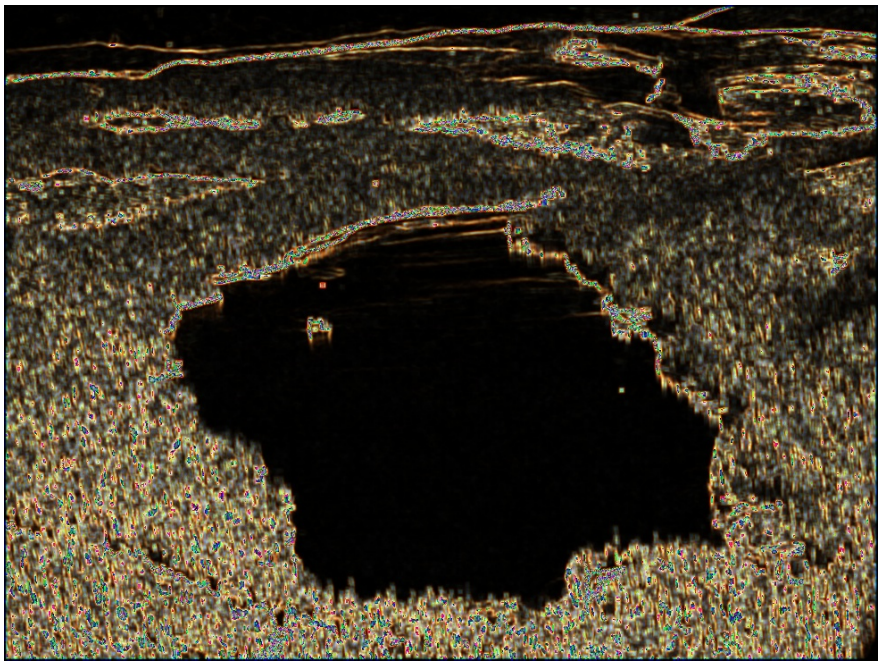
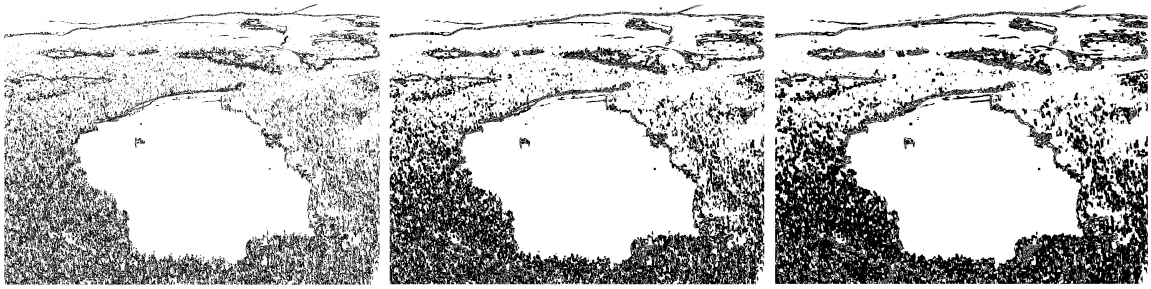


Figure 5: Original image in texture space



(a) Complement of $N = 3$ mask (b) Complement of $N = 5$ mask (c) Complement of $N = 7$ mask

Figure 6: Masks; the final mask is obtained by ANDing all three masks.

Figure 7: Segmented foreground with texture

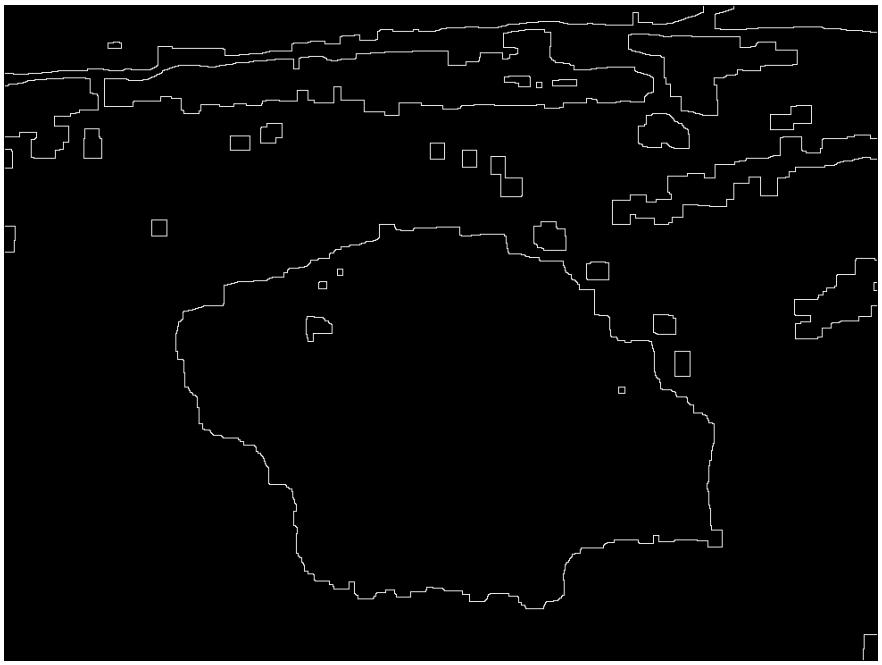
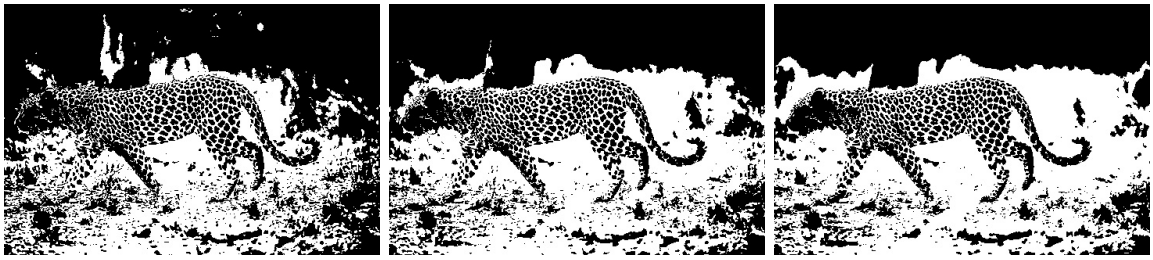


Figure 8: Contour extraction with texture

2.3 Leopard

Figure 9: Original image



(a) B channel mask (b) G channel mask (c) R channel mask

Figure 10: Masks; the final mask is obtained by *AND*ing all three masks.

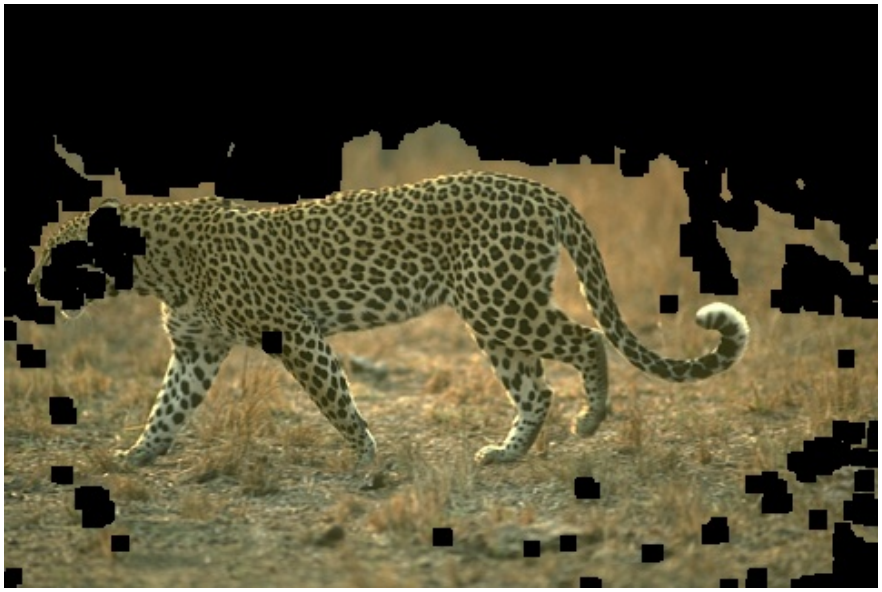


Figure 11: Segmented foreground

Figure 12: Contour extraction

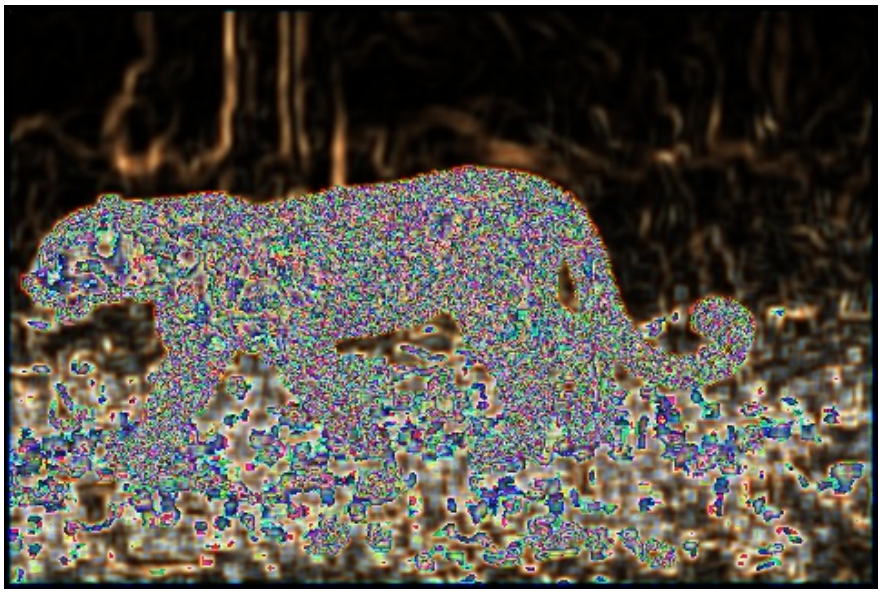
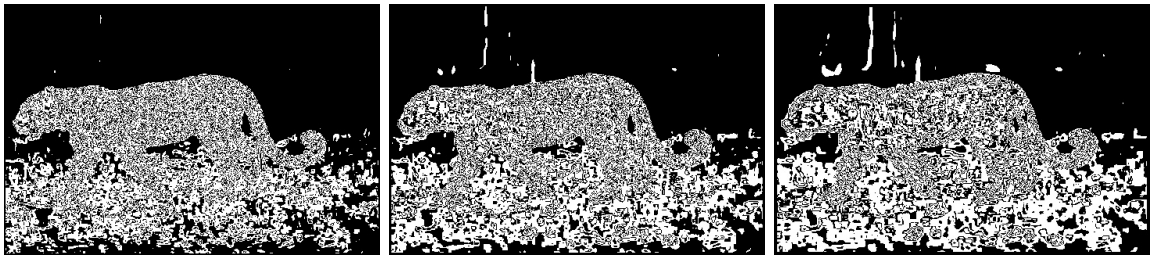


Figure 13: Original image in texture space



(a) Complement of $N = 5$ mask (b) Complement of $N = 7$ mask (c) Complement of $N = 9$ mask

Figure 14: Masks; the final mask is obtained by *AND*ing all three masks.

Figure 15: Segmented foreground with texture

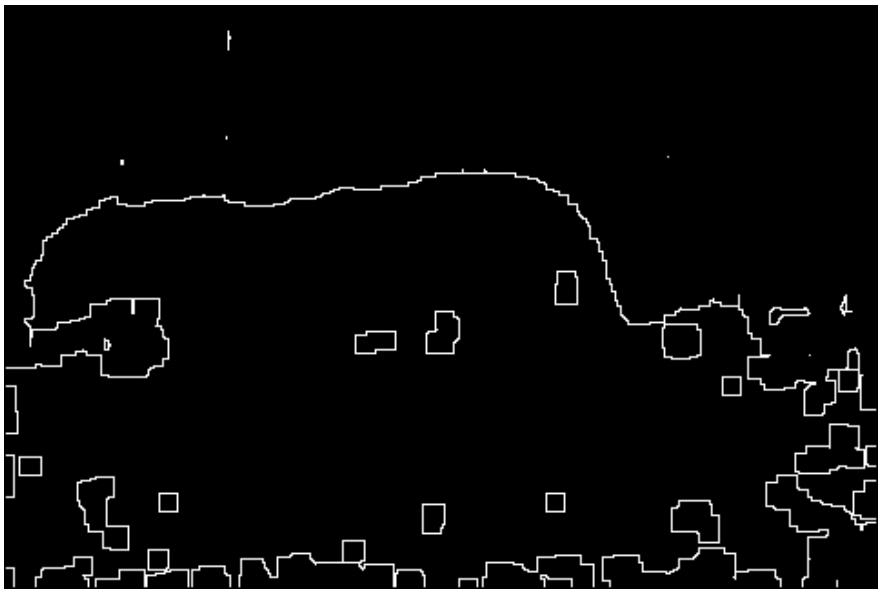


Figure 16: Contour extraction with texture

2.4 Brain

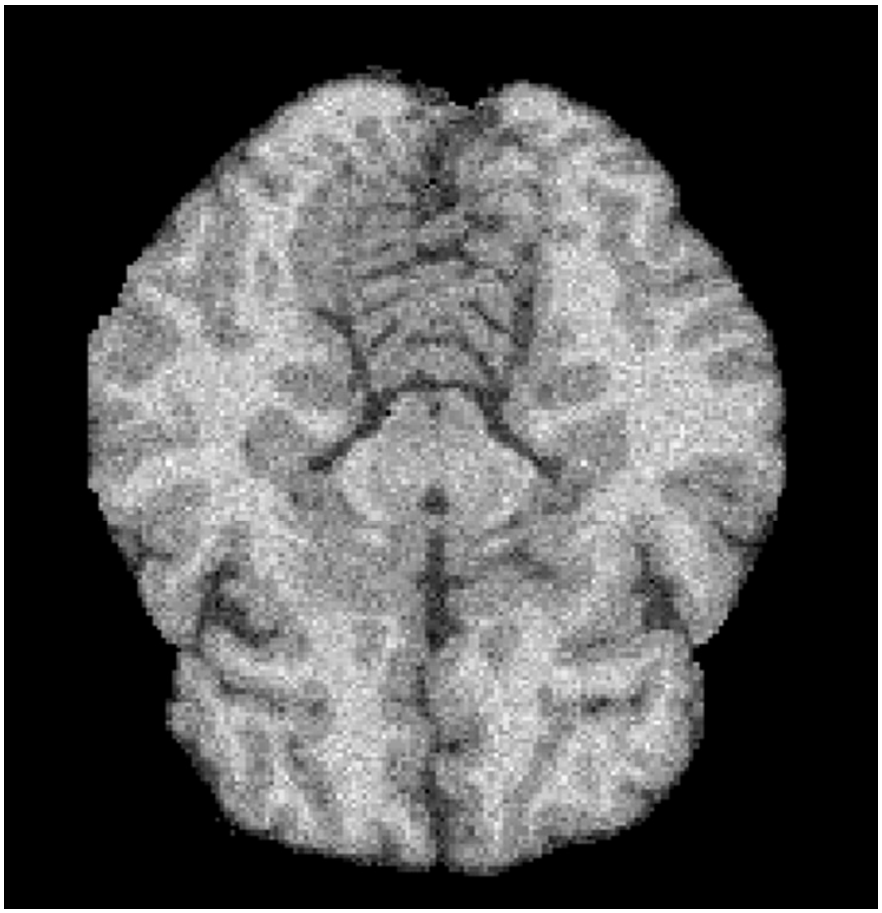
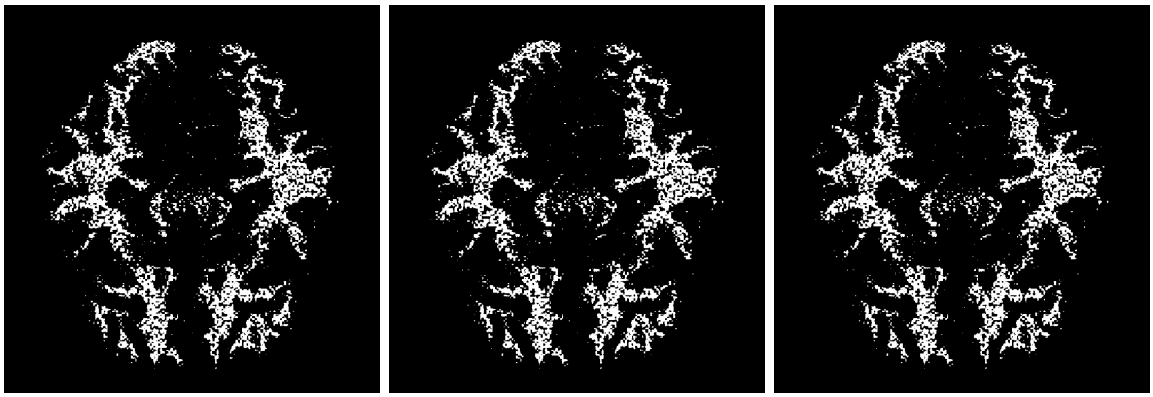


Figure 17: Original image



(a) B channel mask (b) G channel mask (c) R channel mask

Figure 18: Masks; the final mask is obtained by *AND*ing all three masks.

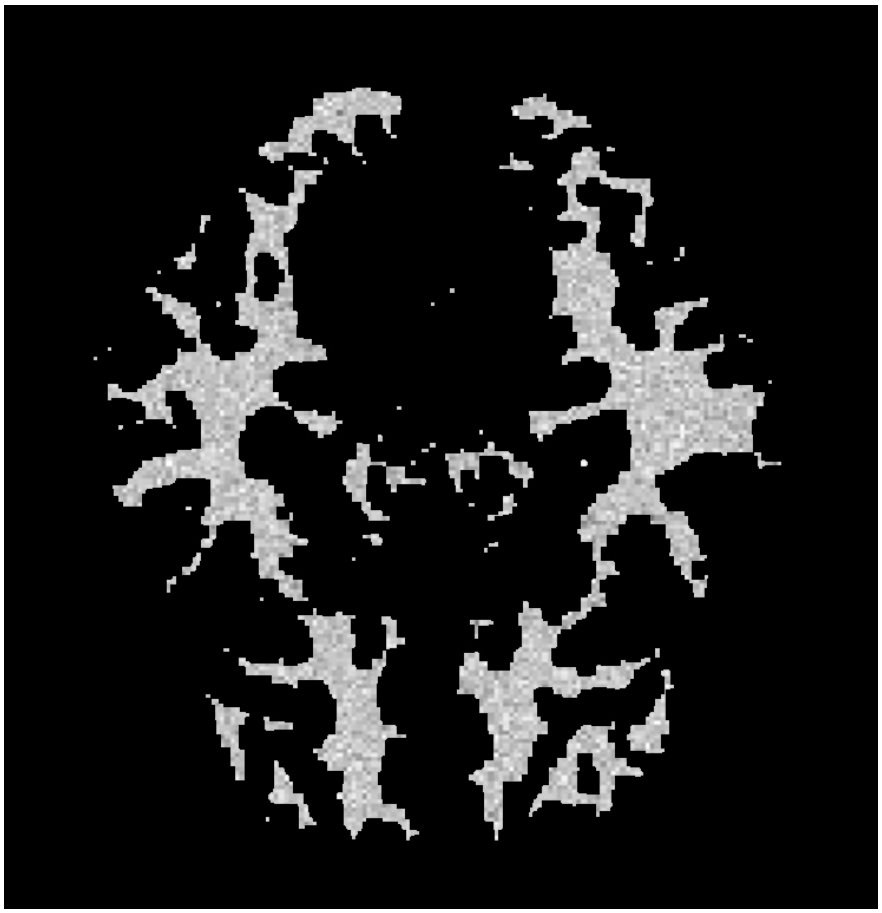


Figure 19: Segmented foreground

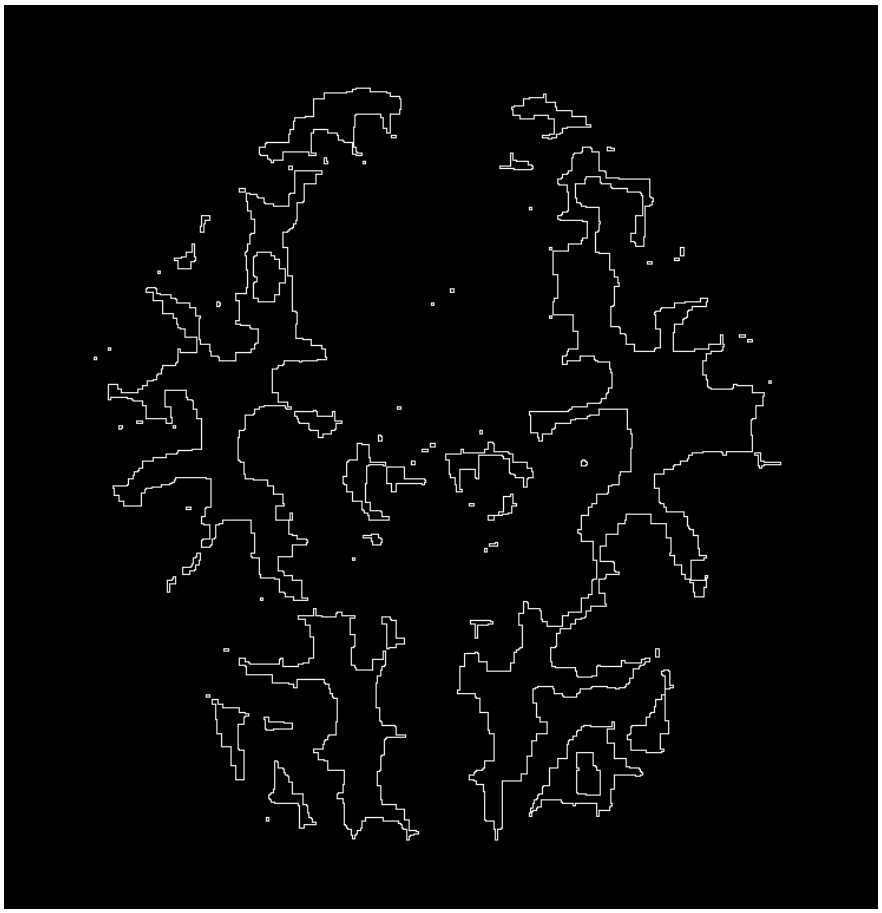


Figure 20: Contour extraction

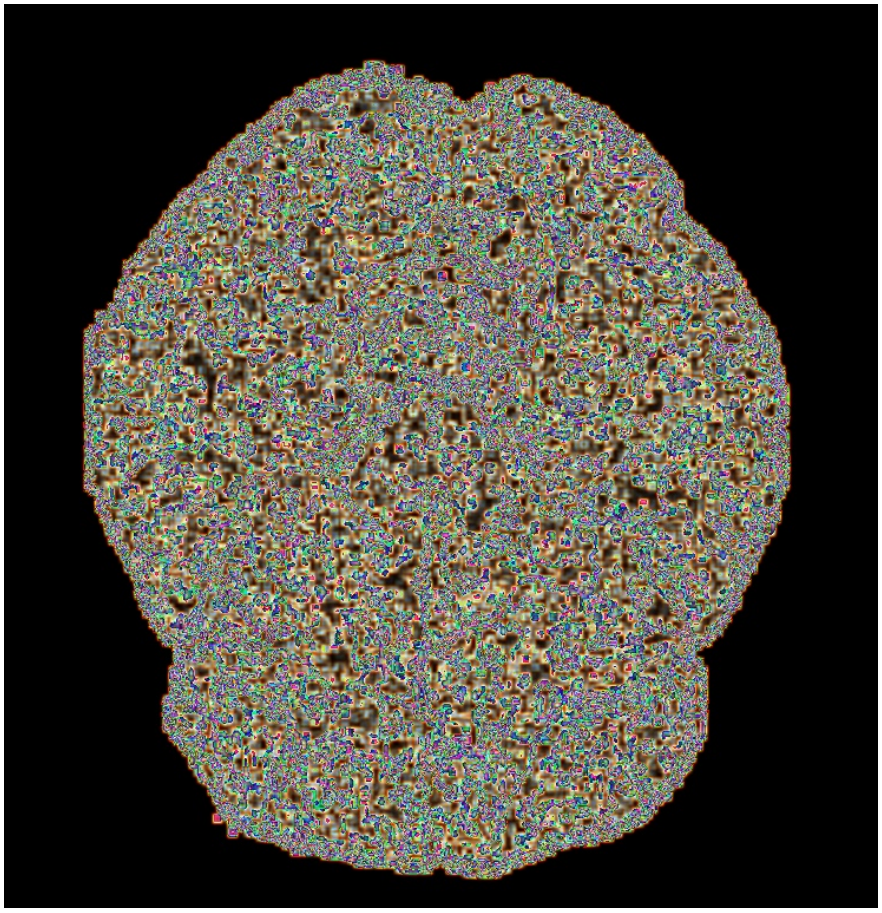
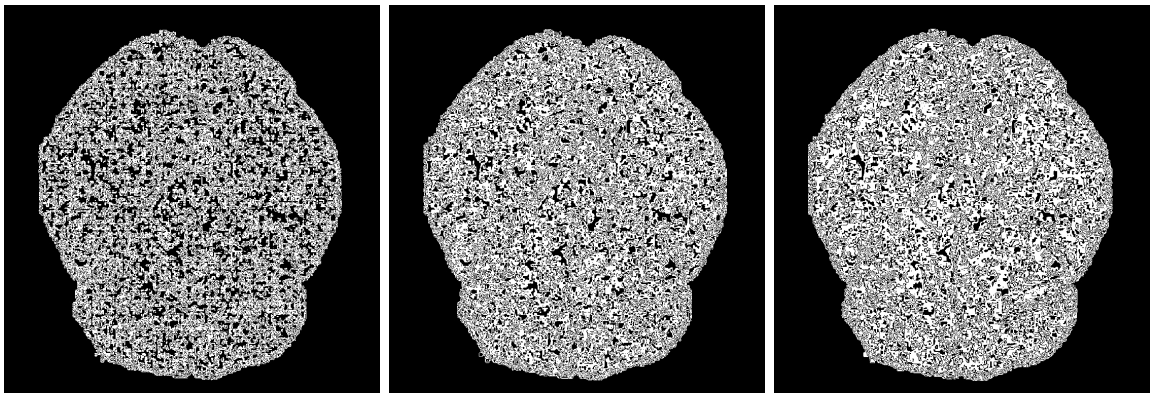


Figure 21: Original image in texture space



(a) Complement of $N = 5$ mask (b) Complement of $N = 7$ mask (c) Complement of $N = 9$ mask

Figure 22: Masks; the final mask is obtained by *AND*ing all three masks.

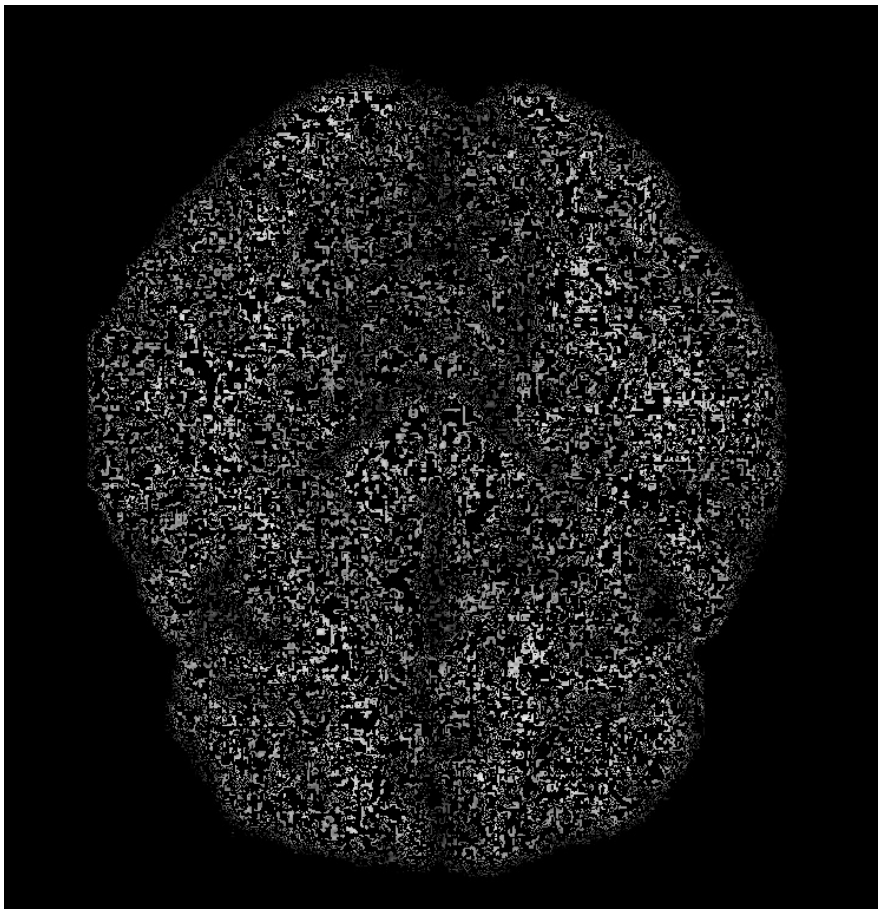


Figure 23: Segmented foreground with texture

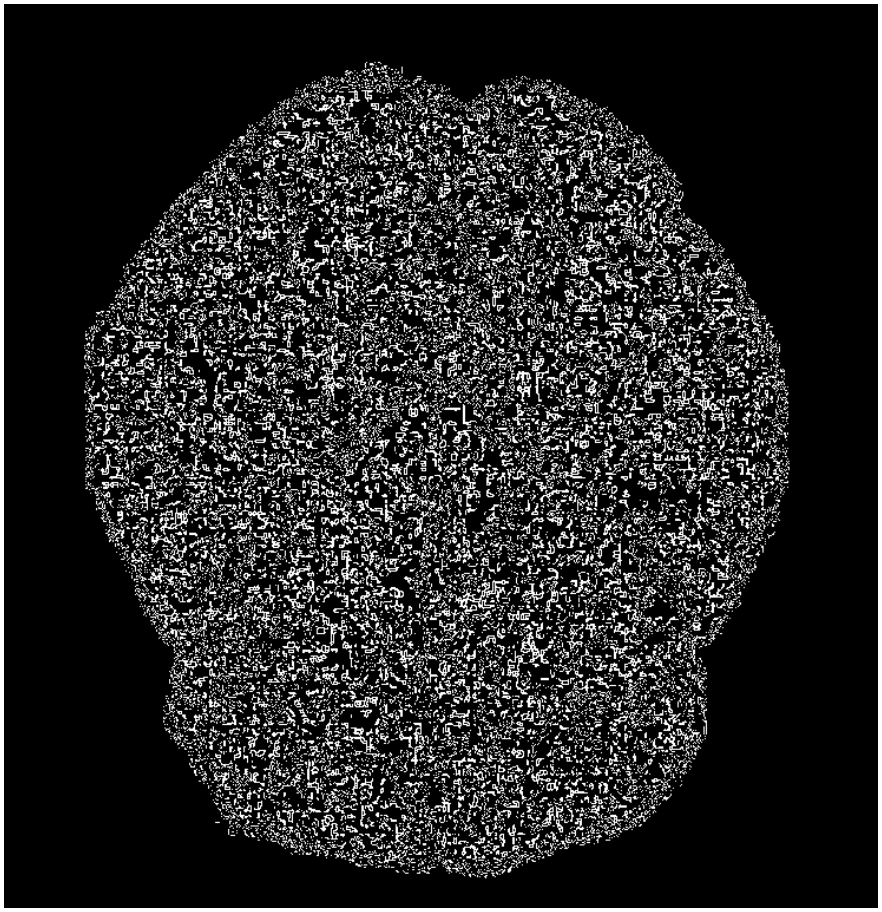


Figure 24: Contour extraction with texture

3 Source Code

3.1 otsu.py

```
#!/usr/bin/python
import numpy as np
import cv2
from matplotlib import pyplot as plt

def otsu_gray(image, nIter):
        '''
                Apply Otsu's algorithm on a gray scale image.
                @image: np.ndarray of input image
                @nIter: int of number of iterations to apply
                @return: np.ndarray of binary mask containing the foreground
        '''
        mask = np.ones(image.shape).astype(np.uint8) * 255
        for i in xrange(nIter):
                hist,_ = np.histogram(image[np.nonzero(mask)], bins=np.arange(257), dens
                ihist = np.multiply(hist, np.arange(256))
                mu_T = np.sum(ihist)
                b_vars = np.zeros(256)
                for k in xrange(256):
                        omega_k = np.sum(hist[:k])
                        mu_k = np.sum(ihist[:k])
                        if omega_k == 0.0 or omega_k == 1.0: continue
                        b_vars[k] = (mu_T*omega_k - mu_k)**2 / omega_k / (1 - omega_k)
                k_star = np.argmax(b_vars)
                _,mask = cv2.threshold(image, k_star, 255, cv2.THRESH_BINARY)
        return mask

def otsu_rgb(image, nIters, invMasks, morphOps):
        '''
                Apply Otsu's algorithm on a RGB image.
                @image: np.ndarray of input image
                @nIters: int of number of iterations to apply
                @invMasks: bool of whether to invert the mask
                @morphOps: int of morphological operation kernel sizes
                @return: np.ndarray of segmented foreground
        '''
        # BGR channels
        channels = cv2.split(image)
        masks = []
        final_mask = np.ones(channels[0].shape).astype(np.uint8) * 255
        for i in range(len(channels)):
```

```
                mask = otsu_gray(channels[i], nIters[i])
                if invMasks[i]:
                        mask = cv2.bitwise_not(mask)
                final_mask = cv2.bitwise_and(final_mask, mask)
                masks.append(mask)
        # Morphological operations to get rid of small regions in background
        kernel = np.ones((morphOps[0],morphOps[0])).astype(np.uint8)
        final_mask = cv2.erode(final_mask, kernel)
        final_mask = cv2.dilate(final_mask, kernel)
        # Morphological operations to fill in holes in foreground
        kernel = np.ones((morphOps[1],morphOps[1])).astype(np.uint8)
        final_mask = cv2.dilate(final_mask, kernel)
        final_mask = cv2.erode(final_mask, kernel)
        return masks, final_mask

def main():
        fpath1 = 'images/lake.jpg'
        color1 = cv2.imread(fpath1)
        final_mask = otsu_rgb(color1, [1,2,2], [0,1,1])


if __name__ == '__main__':
        main()
```

3.2 main.py

```
#!/usr/bin/python
import numpy as np
import cv2
from matplotlib import pyplot as plt
from otsu import otsu_rgb

def get_texture_image(bgr, Ns):
        '''
                Represent the input image in texture space.
                @bgr: np.ndarray of input BGR image
                @Ns: list of int of window sizes
                @return: np.ndarray of image in texture space
        '''
        gray = cv2.cvtColor(bgr, cv2.COLOR_BGR2GRAY)
        texture = np.zeros( (bgr.shape[0], bgr.shape[1], len(Ns)) )
        for i, N in enumerate(Ns):
                w_h = int(N/2)
                for r in xrange(w_h, bgr.shape[0]-w_h):
                        for c in xrange(w_h, bgr.shape[1]-w_h):
```

```
                                var = np.var( gray[r-w_h:r+w_h+1, c-w_h:c+w_h+1] )
                                texture[r,c,i] = var
        return texture.astype(np.uint8)

def get_contour(mask):
        '''
                Given an binary image, return a binary image that only contains the co
                @mask: np.ndarray of binary input image, i.e. 0 or 255
                @return: np.ndarray of binary contour image
        '''
        contour = np.zeros(mask.shape).astype(np.uint8)
        w_h = 1
        for r in xrange(w_h, mask.shape[0]-w_h):
                for c in xrange(w_h, mask.shape[1]-w_h):
                        if mask[r,c] == 0:
                                continue
                        if np.min( mask[r-w_h:r+w_h+1, c-w_h:c+w_h+1] ) == 0:
                                contour[r,c] = 255
        return contour

def main():
        name = 'leopard'
        useTexture = True
        fpath = './images/' + name + '.jpg'
        if name == 'lake':
                # Lake
                nIters = [1,2,2]
                invMasks = [0,1,1]
                morphOps = [17,5]
                color = cv2.imread(fpath)
                if useTexture:
                        Ns = [3,5,7]
                        nIters = [1,1,1]
                        invMasks = [1,1,1]
                        morphOps = [17,5]
                        texture = get_texture_image(color, Ns)
                        name = name + '_t'
        elif name == 'leopard':
                # Leopard
                nIters = [1,1,1]
                invMasks = [0,0,0]
                morphOps = [1,9]
                color = cv2.imread(fpath)
                if useTexture:
                        Ns = [5,7,9]
```

```
                nIters = [1,1,1]
                invMasks = [0,0,0]
                morphOps = [1,9]
                texture = get_texture_image(color, Ns)
                name = name + '_t'
elif name == 'brain':
        # Brain
        nIters = [3,3,3]
        invMasks = [0,0,0]
        morphOps = [3,11]
        color = cv2.imread(fpath)
        if useTexture:
                Ns = [5,7,9]
                nIters = [1,1,1]
                invMasks = [0,0,0]
                morphOps = [1,1]
                texture = get_texture_image(color, Ns)
                name = name + '_t'

if useTexture:
        image = texture
else:
        image = color
channels = cv2.split(image)
masks, final_mask = otsu_rgb(image, nIters, invMasks, morphOps)
foreground = cv2.bitwise_and(color, cv2.cvtColor(final_mask, cv2.COLOR_GRAY2BGR))
contour = get_contour(final_mask)
# Save the images
cv2.imwrite('./images/' + name + '_foreground.jpg', foreground)
cv2.imwrite('./images/' + name + '_finalmask.jpg', final_mask)
cv2.imwrite('./images/' + name + '_contour.jpg', contour)
for i in range(len(channels)):
        cv2.imwrite('./images/' + name + '_mask_' + str(i) + '.jpg', masks[i])
if useTexture:
        cv2.imwrite('./images/' + name + '_texture.jpg', texture)
# Plots
plt.imshow(cv2.cvtColor(foreground, cv2.COLOR_BGR2RGB))
fig, axes = plt.subplots(2,3)
for i in range(len(channels)):
        axes[0,i].set_aspect('equal')
        axes[0,i].imshow(channels[i], cmap='gray')
        axes[1,i].set_aspect('equal')
        axes[1,i].imshow(masks[i], cmap='gray')
plt.figure()
plt.imshow(final_mask, cmap='gray')
```

```
        plt.figure()
        plt.imshow(contour, cmap='gray')
        if useTexture:
                plt.figure()
                plt.imshow(texture)
        plt.show()

if __name__ == '__main__':
        main()

        # color = cv2.cvtColor(color, cv2.COLOR_BGR2HSV)
        # masks, final_mask = otsu_rgb(color, [2,1,1], [0,0,0])
        # cv2.imwrite('./images/' + name + '_foreground.jpg', cv2.cvtColor(foreground,
```