Image Processing Assignment 2

Introduction

- Goal:
  - ■ Experiment with one of several image segmentation techniques mentioned in the class. The options are listed below:
    1. Canny edge detector
    2. Watershed algorithm
    3. Hough transform
    4. The SLIC super-pixel algorithm

    → I choose Watershed algorithm
  - ■ Experiment with 4~8 images in this assignment
  - ■ You CAN NOT use toolbox/library functions
- Environment: Python 3.8.8
- Code: https://github.com/maikurufeza/NCTU-Image-Processing-2021/blob/main/Assignment/Assignment2_final_version.ipynb

Implemented method

- Watershed algorithm

  A. Watershed transformation

  B. Watershed transformation with region merging

  C. Watershed transformation with automatically marker

  D. Watershed transformation with interactive marker

Organization

- Section 1 (Experiments): I show all the method I practice and its effect.
- Section 2 (Observation and Discussions):

  I show some my observation. And because the problem of over-segmentation, I try my best to improve the result and discussion how I improve.
- Section 3 (Code Analysis):

  This part shows the code of the method above. Note that the previous methods of code in Assignment 1 do not show in this part.

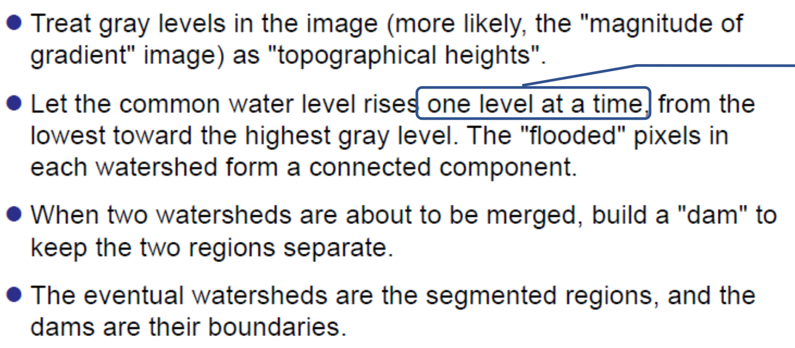
※ Note: Because I can't put animations in the document. The attachments of watershed flooding animations are in the drive: https://drive.google.com/drive/folders/1jg-G-uEirg04oLMWLXeuaxw82QeznoNZ?usp=sharing

Section 1: Experiments

※ Note: All image is preprocessed by averaging filter for reducing over-segmentation.

I. Watershed transformation

Method:



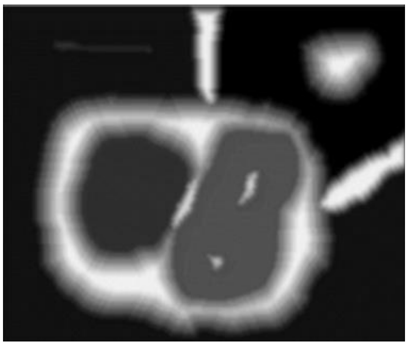
- Treat gray levels in the image (more likely, the "magnitude of gradient" image) as "topographical heights".
- Let the common water level rises one level at a time from the lowest toward the highest gray level. The "flooded" pixels in each watershed form a connected component.
- When two watersheds are about to be merged, build a "dam" to keep the two regions separate.
- The eventual watersheds are the segmented regions, and the dams are their boundaries.

can be more than one level

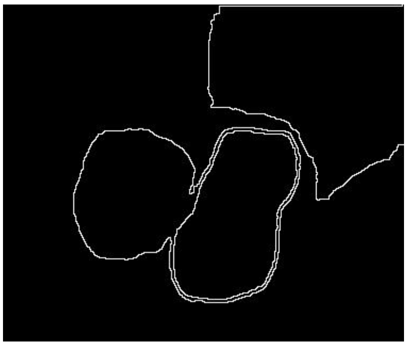
Level_jump: How many levels rises at a time

Experiment:

Original image



Segmentation after watershed



Parameter:

Level_jump: 27

Watershed Animation:
See the attachment of ‘example.gif’

Result: The ‘dam’ forms the segmentation of the image.

II. Region merging:

Method:

(1). Form initial regions in the image.

(2). Build a regions adjacency graph (RAG).

(3). For each region do:

(3.1). Consider its adjacent region and test to see if they are similar.

(3.2). For regions that are similar, merge them and modify the RAG.

(4). Repeat step 3 until no regions are merged.

Similarity_thresh: How close mean intensities will

Compare their mean intensities

Experiment:

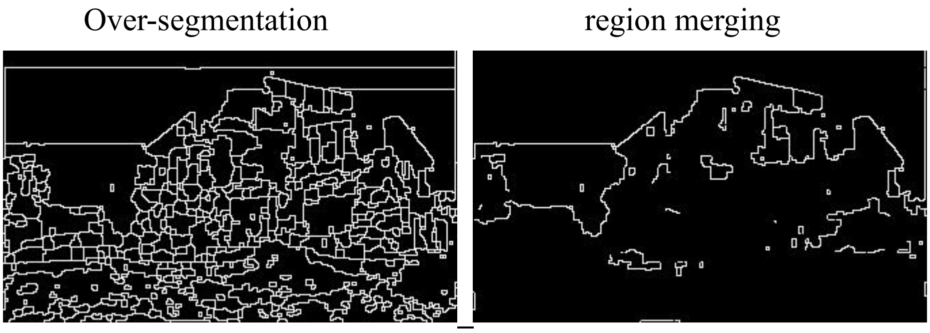
Original image

Parameter:

Level_jump: 10

Similarity_thresh: 20

Watershed Animation:
See the attachment of ‘house.gif’


Over-segmentation
region merging

Result: Region merging can reduce over-segmentation. But it may lead to under-segmentation.

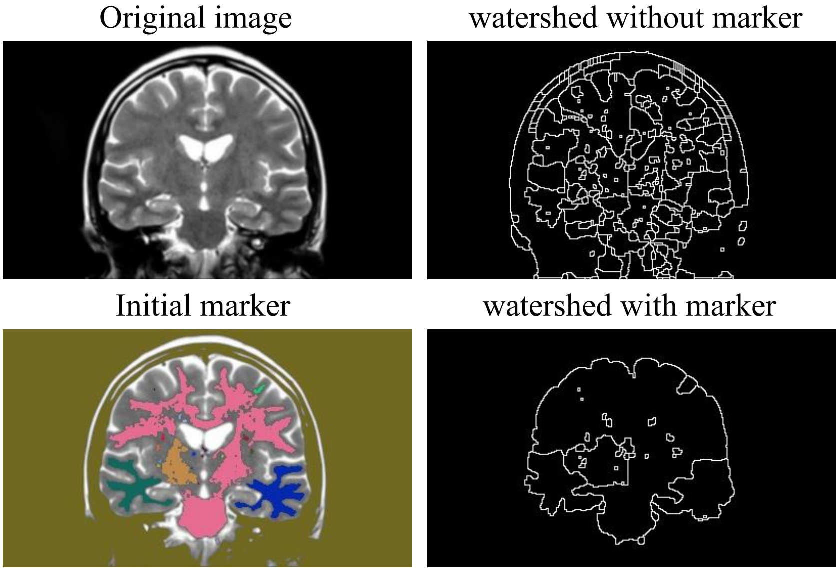
III. Watershed with automatically marker

Method:

$Marker\_thresh$

Choose the region which is lower than the threshold to be the marker.

Experiment:


Original image
watershed without marker
Initial marker
watershed with marker

Colors for different markers

Parameter:
$Level\_jump$: 10
$Marker\_thresh$: 100

Watershed Animation:
See the attachment of ‘brain_mri.gif’

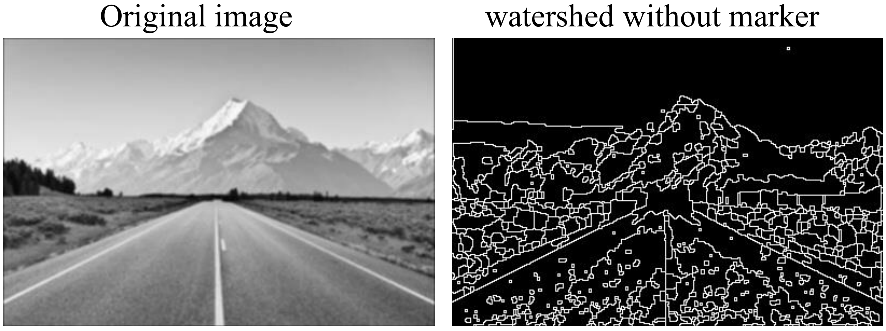
Result: Automatically marker can reduce over-segmentation.

IV. Watershed with interactively marker

Method:

The marker is drawn by the user.

Experiment:

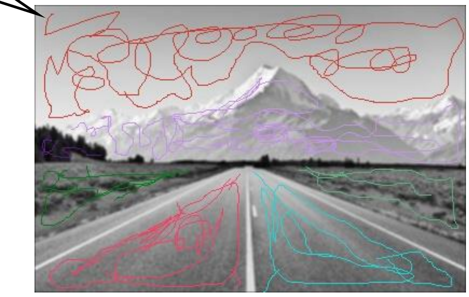

Original image
watershed without marker

Parameter:
$Level\_jump$: 10

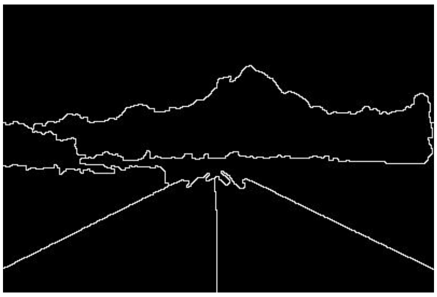
Watershed Animation:
See the attachment of ‘scene.gif’

Colors for different markers

Drawn marker



watershed with marker



Result: Interactively marker can reduce over-segmentation

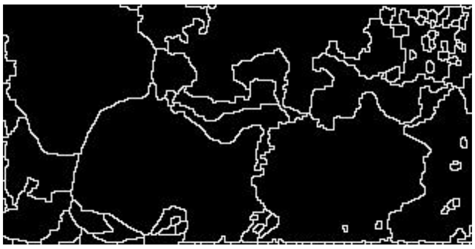
Section 2: Observation and Discussion

※ Note: In order to reduce over-segmentation, the following watershed implemented with automatically marker for convenience if I don't mention.

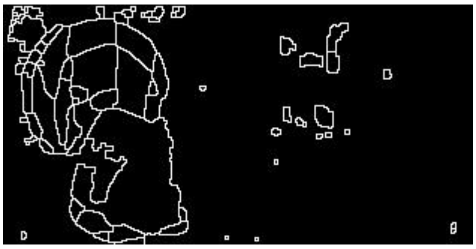
1. What if I do watershed on color image? i.e., do on blue, green, red channel.

Original image

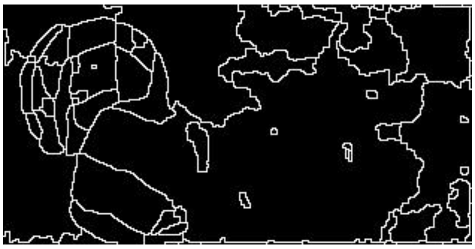
watershed



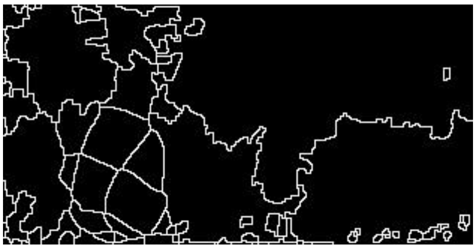
Watershed on blue channel



Watershed on green channel



Watershed on red channel



Parameter:

$Level\_jump$: 10

$Marker\_thresh$: 64

Watershed Animation:

See the attachment of 'balls'

It seems that watershed on color image is not helpful.

2. The less $Level\_jump$ is, the more over-segmentation is.

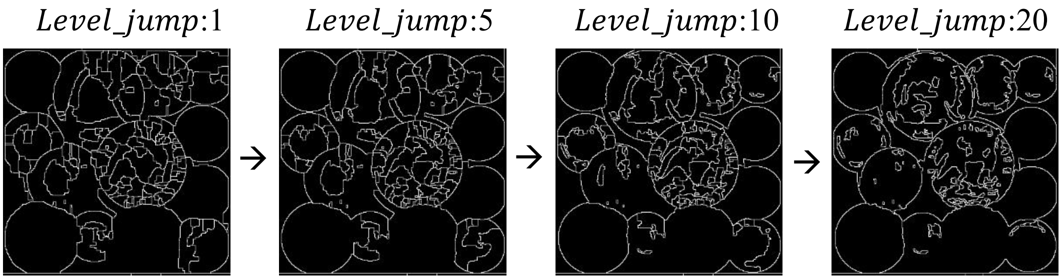
← original image

Parameter:

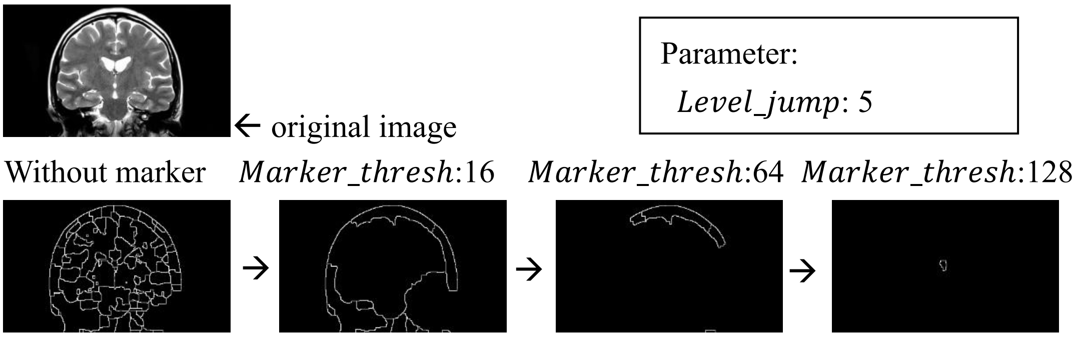
$Marker\_thresh$: 128

Watershed Animation:

See the attachment of 'coins'

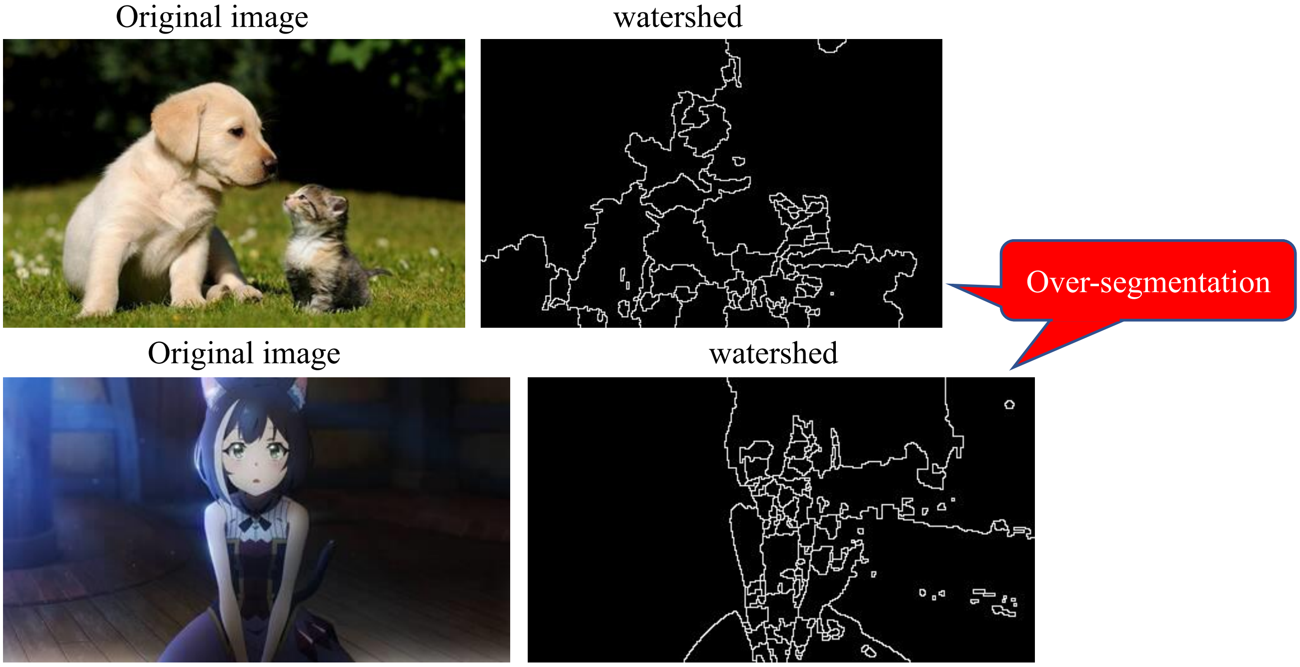


3. The more $Marker\_thresh$ is, the more under-segmentation is



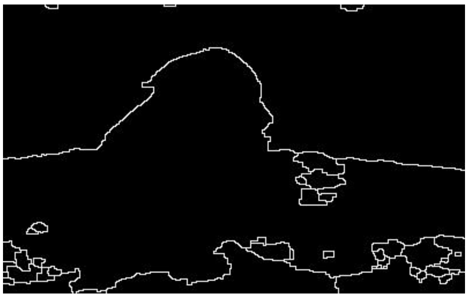
Parameter:
$Level\_jump$: 5

4. Watershed transformation is easy to get over-segmentation image. Especially more complex images such as pets, people. The example is below. We are hard to see the shape of pets and person in the watershed segmentation.

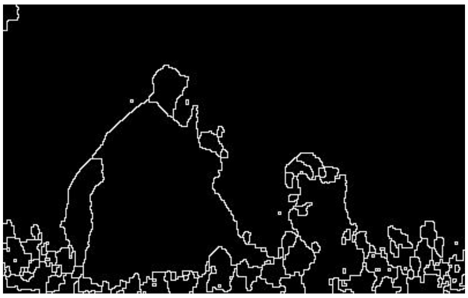


Therefore, I think I can do some image preprocessing by computing gradients. I try bottom sobel filter and right sobel filter before watershed transformation. And the results show below.

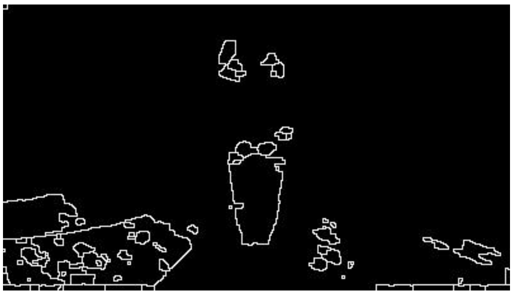
Watershed after bottom sobel



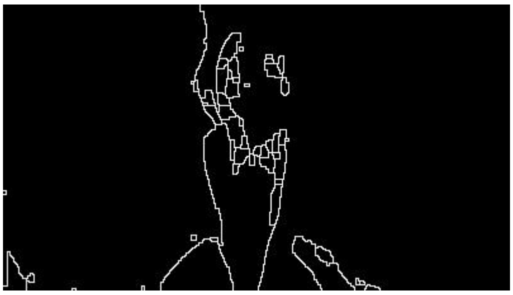
Watershed after right sobel



Watershed after bottom sobel



Watershed after right sobel



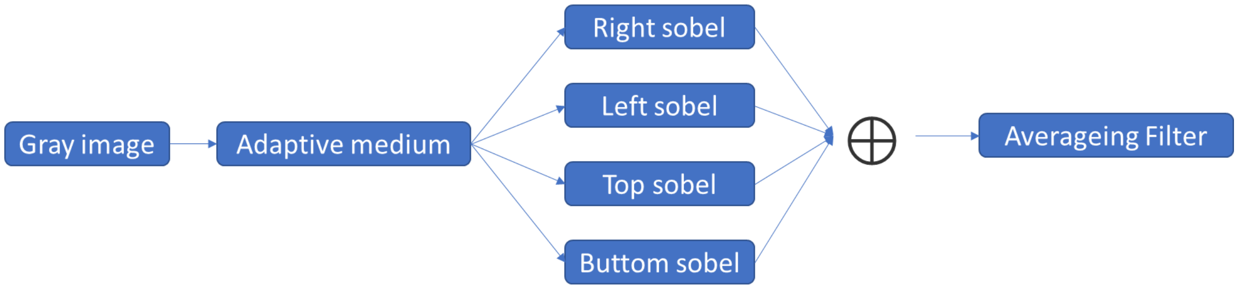
I found there are two problems:

1. Bottom sobel filter tends to get horizontal segmentation, right sobel filter tends to get vertical segmentation. Sobel filter just can get certain direction of segmentation. I have tried Laplace filter. But it didn't improve.
2. There are a lot of small regions in segmentation. I think this is cause by the noise in the image

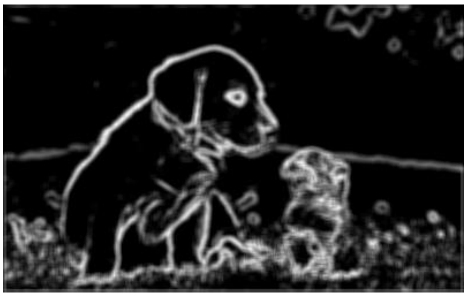
To solve problem 1, I calculate all direction of sobel and sum all direction of sobel.

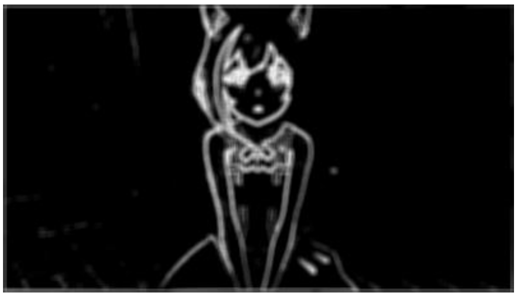
To solve problem 2, I use adaptive medium filter to reduce noise.

Therefore, I did the preprocessing below to improve over-segmentation.

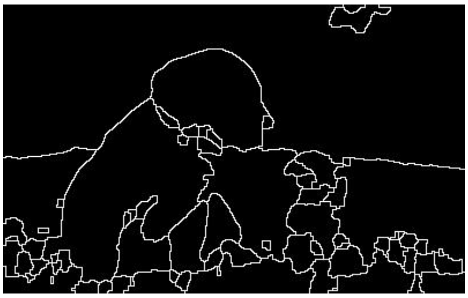


After the preprocessing above, I got the images with less noise and all direction of edge. The result is shown below.





Then I put the preprocessed image into watershed. I got better segmentation. The result segmentation is shown below.



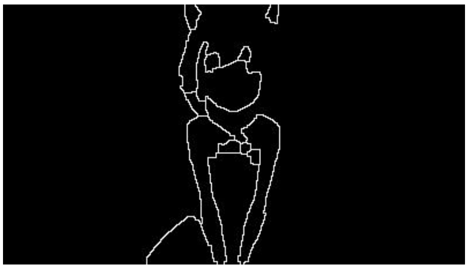
Parameter:

$Level\_jump$: 10

$Marker\_thresh$: 32

Watershed Animation:

See the attachment of 'pets.gif'



Parameter:

$Level\_jump$: 5

$Marker\_thresh$: 16

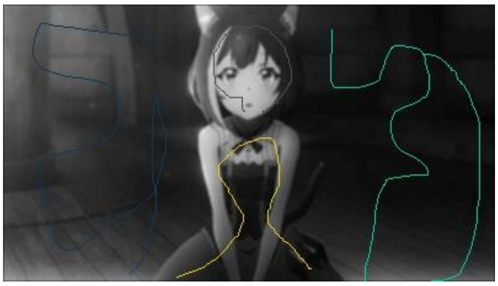
Watershed Animation:

See the attachment of 'Karyl.gif'

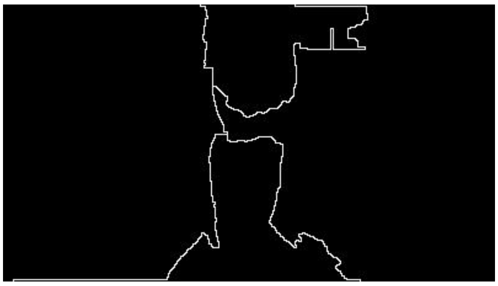
In the image of pets, we can see the shape of the dog! And we can see the shape of cat slightly. In the image of the person, we can also see the shape of the person!

5. How does the drawn marker affect the segmentation? I make comparison between drawing markers casually and carefully.

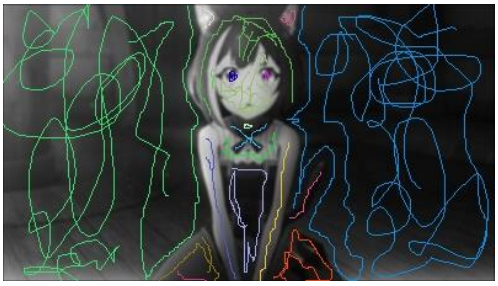
Casually markers



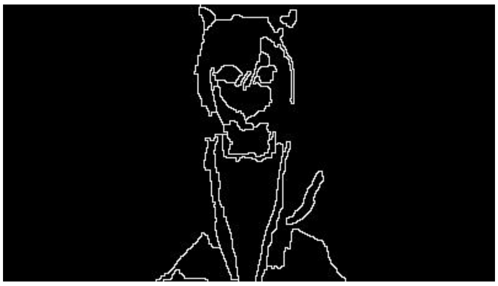
Watershed with casually markers



Carefully markers

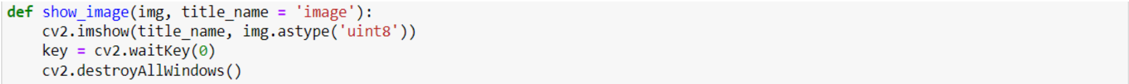


Watershed with carefully markers

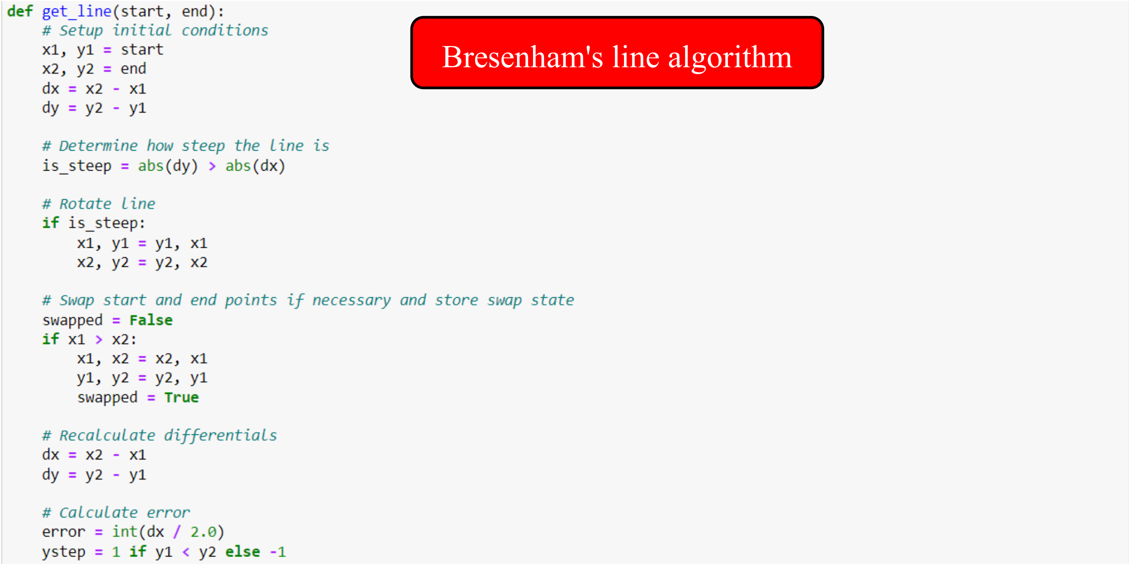


Although carefully markers get better segmentation, but it is very time-consuming. (Because the user must spend time to draw markers) And the improvement is not better than imaging preprocessing mention in 4.

Code Analysis

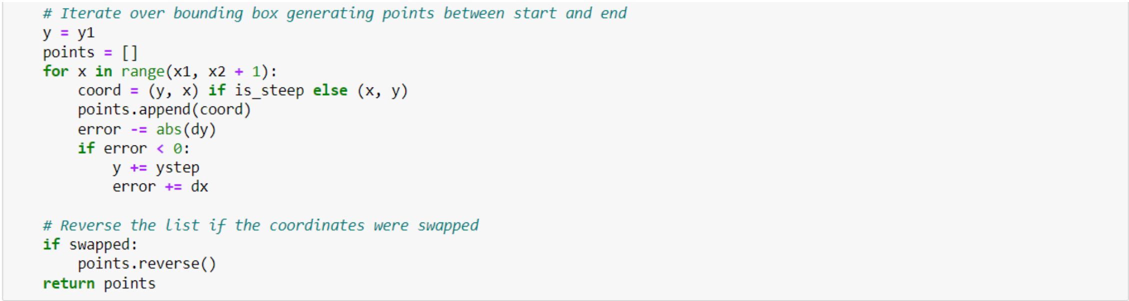


```python
def show_image(img, title_name = 'image'):
    cv2.imshow(title_name, img.astype('uint8'))
    key = cv2.waitKey(0)
    cv2.destroyAllWindows()
```

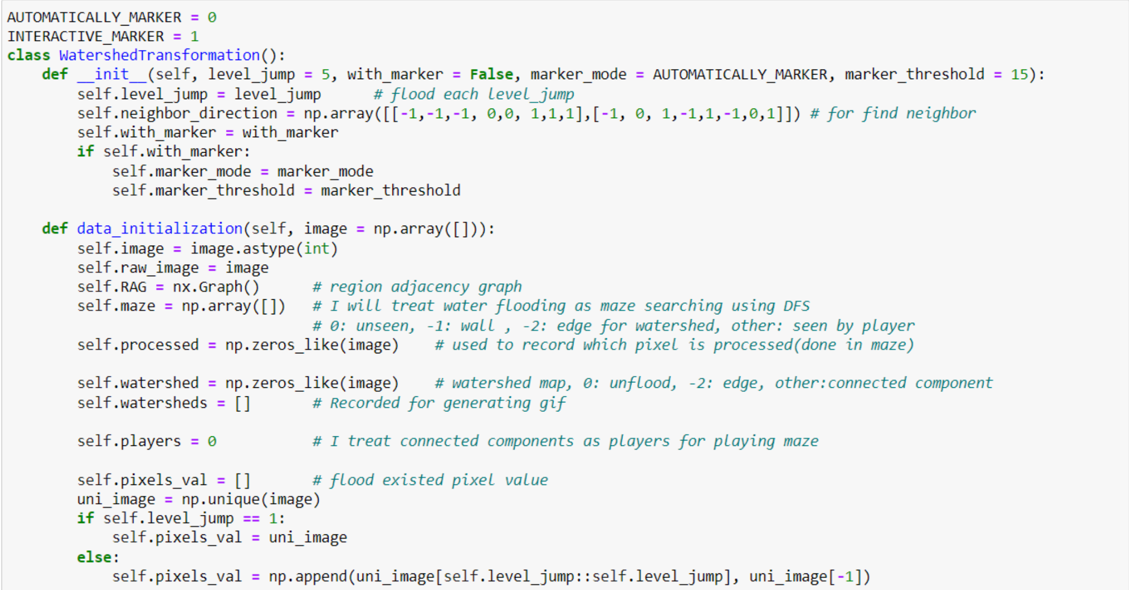


Bresenham's line algorithm

```python
def get_line(start, end):
    # Setup initial conditions
    x1, y1 = start
    x2, y2 = end
    dx = x2 - x1
    dy = y2 - y1

    # Determine how steep the line is
    is_steep = abs(dy) > abs(dx)

    # Rotate line
    if is_steep:
        x1, y1 = y1, x1
        x2, y2 = y2, x2

    # Swap start and end points if necessary and store swap state
    swapped = False
    if x1 > x2:
        x1, x2 = x2, x1
        y1, y2 = y2, y1
        swapped = True

    # Recalculate differentials
    dx = x2 - x1
    dy = y2 - y1

    # Calculate error
    error = int(dx / 2.0)
    ystep = 1 if y1 < y2 else -1
```



```python
    # Iterate over bounding box generating points between start and end
    y = y1
    points = []
    for x in range(x1, x2 + 1):
        coord = (y, x) if is_steep else (x, y)
        points.append(coord)
        error -= abs(dy)
        if error < 0:
            y += ystep
            error += dx

    # Reverse the list if the coordinates were swapped
    if swapped:
        points.reverse()
    return points
```

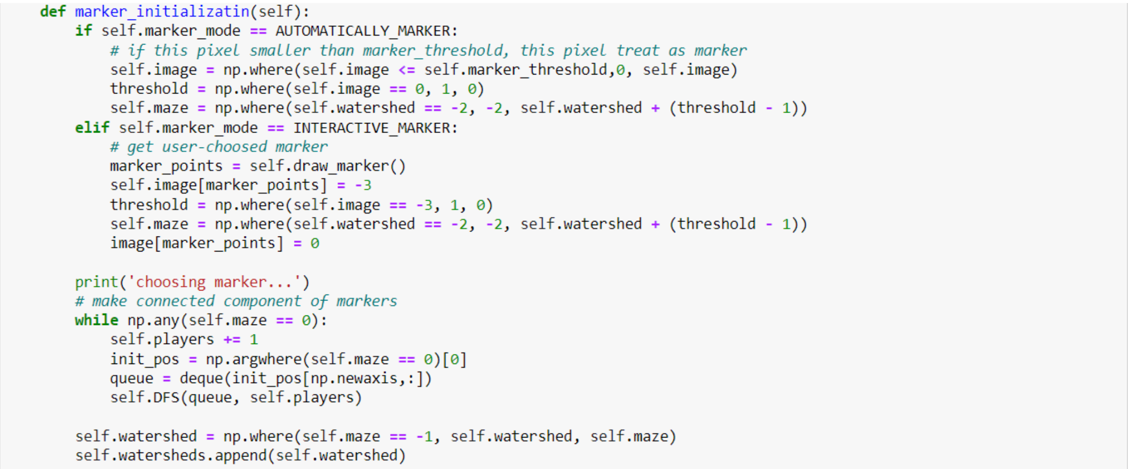


```python
AUTOMATICALLY_MARKER = 0
INTERACTIVE_MARKER = 1
class WatershedTransformation():
    def __init__(self, level_jump = 5, with_marker = False, marker_mode = AUTOMATICALLY_MARKER, marker_threshold = 15):
        self.level_jump = level_jump      # flood each level_jump
        self.neighbor_direction = np.array([[-1,-1,-1, 0,0, 1,1,1],[-1, 0, 1,-1,1,-1,0,1]]) # for find neighbor
        self.with_marker = with_marker
        if self.with_marker:
            self.marker_mode = marker_mode
            self.marker_threshold = marker_threshold

    def data_initialization(self, image = np.array([])):
        self.image = image.astype(int)
        self.raw_image = image
        self.RAG = nx.Graph()       # region adjacency graph
        self.maze = np.array([])    # I will treat water flooding as maze searching using DFS
                                    # 0: unseen, -1: wall , -2: edge for watershed, other: seen by player
        self.processed = np.zeros_like(image)    # used to record which pixel is processed(done in maze)

        self.watershed = np.zeros_like(image)    # watershed map, 0: unflood, -2: edge, other:connected component
        self.watersheds = []        # Recorded for generating gif

        self.players = 0            # I treat connected components as players for playing maze

        self.pixels_val = []        # flood existed pixel value
        uni_image = np.unique(image)
        if self.level_jump == 1:
            self.pixels_val = uni_image
        else:
            self.pixels_val = np.append(uni_image[self.level_jump::self.level_jump], uni_image[-1])
    def get_neightbor_indices(self, index, player):
        neighbor_indices = []      # Record the neighbor of index
        neightbor_all_zero = True  # check the pixel is processed totally
        for d in range(8):
            neighbor_index = self.neighbor_direction[:,d] + index

            # skip those neighbor which can't visit
            if np.any(neighbor_index<0) or np.any(neighbor_index>=self.image.shape) or \
                self.maze[neighbor_index[0], neighbor_index[1]] == -2 or \
                self.processed[neighbor_index[0], neighbor_index[1]] == 1 or \
                self.maze[neighbor_index[0], neighbor_index[1]] == player:
                continue
            # neighbor have not been able to processed
            elif self.maze[neighbor_index[0], neighbor_index[1]] == -1:
                neightbor_all_zero = False
            # collect neighbors which can access
            elif self.maze[neighbor_index[0], neighbor_index[1]] == 0:
                neighbor_indices.append(neighbor_index)
            # touch other's component's area
            elif self.maze[neighbor_index[0], neighbor_index[1]] != player:
                self.maze[index[0], index[1]] = -2       # record as the edge in image
                self.processed[index[0], index[1]] = 1    # edge can't process, so record as processed

                # generate edge in RAG
                if not self.RAG.has_edge(player, self.maze[neighbor_index[0], neighbor_index[1]]):
                    self.RAG.add_edge(player, self.maze[neighbor_index[0], neighbor_index[1]])
                    self.RAG[player][self.maze[neighbor_index[0], neighbor_index[1]]]['bound'] = []
                # record pixel of edge
                self.RAG[player][self.maze[neighbor_index[0], neighbor_index[1]]]['bound'].append(index)

                return []
```

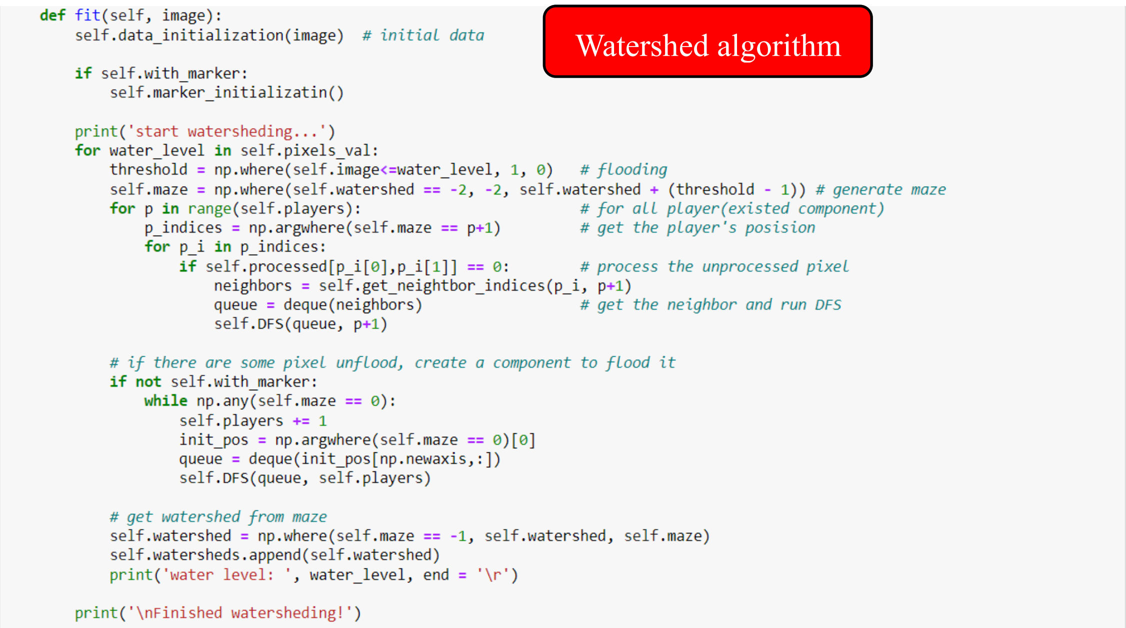
```
        if neightbor_all_zero:
            self.processed[index[0], index[1]] = 1

        if len(neighbor_indices) != 0:
            neighbor_indices = np.stack(neighbor_indices,axis = 0)
        return neighbor_indices

    def DFS(self, queue, player):
        # iterate depth first search (using queue)
        while queue:
            p_out = queue.popleft()
            if self.maze[p_out[0],p_out[1]] == 0:
                self.maze[p_out[0],p_out[1]] = player

                neighbors_p_out = self.get_neightbor_indices(p_out, player)
                for npo in neighbors_p_out:
                    queue.append(npo)
```

```
    def draw_marker(self):
        print('painting marker...')
        global pt1_x,pt1_y,drawing
        drawing = False      # true if mouse is pressed
        pt1_x , pt1_y = None , None
        painting_point = []

        img = np.copy(self.image).astype('uint8')

        # mouse callback function
        def line_drawing(event,x,y,flags,param):
            global pt1_x,pt1_y,drawing
            if event==cv2.EVENT_LBUTTONDOWN:
                drawing=True
                pt1_x,pt1_y=x,y
            elif event==cv2.EVENT_MOUSEMOVE:
                if drawing==True:
                    cv2.line(img,(pt1_x,pt1_y),(x,y),color=(255,255,255),thickness=1)
                    line_point = get_line((pt1_x,pt1_y),(x,y))[:-1]
                    painting_point.extend(line_point)    # record the point user have drawn
                    pt1_x,pt1_y=x,y
            elif event==cv2.EVENT_LBUTTONUP:
                drawing=False
                cv2.line(img,(pt1_x,pt1_y),(x,y),color=(255,255,255),thickness=1)
                line_point = get_line((pt1_x,pt1_y),(x,y))
                painting_point.extend(line_point)        # record the point user have drawn

        cv2.namedWindow('(press esc exit) draw markers')
        cv2.setMouseCallback('(press esc exit) draw markers',line_drawing)
```

```
        # call window
        while(1):
            cv2.imshow('(press esc exit) draw markers',img)
            if cv2.waitKey(1) & 0xFF == 27:    #press esc exit
                break
        cv2.destroyAllWindows()
        self.drawing_image = img

        # delete out-of-bound point
        paint_i = [p[1] for p in painting_point if p[1] < img.shape[0] and p[0] < img.shape[1]\
                  and p[1] >= 0 and p[0] >= 0]
        paint_j = [p[0] for p in painting_point if p[1] < img.shape[0] and p[0] < img.shape[1]\
                  and p[1] >= 0 and p[0] >= 0]
        return (paint_i, paint_j)
```

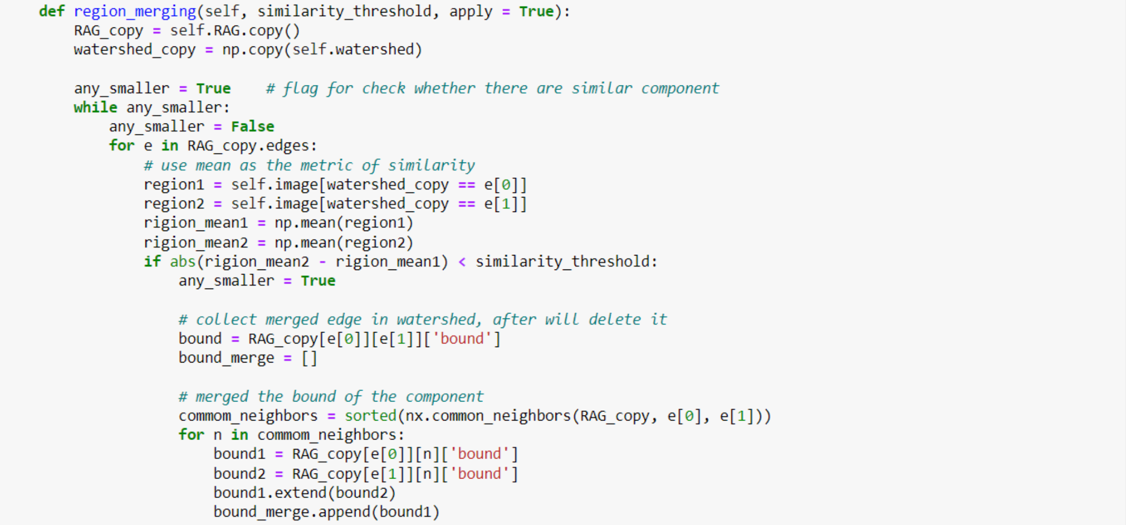


```
    def marker_initializatin(self):
        if self.marker_mode == AUTOMATICALLY_MARKER:
            # if this pixel smaller than marker_threshold, this pixel treat as marker
            self.image = np.where(self.image <= self.marker_threshold,0, self.image)
            threshold = np.where(self.image == 0, 1, 0)
            self.maze = np.where(self.watershed == -2, -2, self.watershed + (threshold - 1))
        elif self.marker_mode == INTERACTIVE_MARKER:
            # get user-choosed marker
            marker_points = self.draw_marker()
            self.image[marker_points] = -3
            threshold = np.where(self.image == -3, 1, 0)
            self.maze = np.where(self.watershed == -2, -2, self.watershed + (threshold - 1))
            image[marker_points] = 0

        print('choosing marker...')
        # make connected component of markers
        while np.any(self.maze == 0):
            self.players += 1
            init_pos = np.argwhere(self.maze == 0)[0]
            queue = deque(init_pos[np.newaxis,:])
            self.DFS(queue, self.players)

        self.watershed = np.where(self.maze == -1, self.watershed, self.maze)
        self.watersheds.append(self.watershed)
```



Watershed algorithm

```
    def fit(self, image):
        self.data_initialization(image)  # initial data

        if self.with_marker:
            self.marker_initializatin()

        print('start watersheding...')
        for water_level in self.pixels_val:
            threshold = np.where(self.image<=water_level, 1, 0)   # flooding
            self.maze = np.where(self.watershed == -2, -2, self.watershed + (threshold - 1)) # generate maze
            for p in range(self.players):                         # for all player(existed component)
                p_indices = np.argwhere(self.maze == p+1)         # get the player's posision
                for p_i in p_indices:
                    if self.processed[p_i[0],p_i[1]] == 0:        # process the unprocessed pixel
                        neighbors = self.get_neightbor_indices(p_i, p+1)
                        queue = deque(neighbors)                  # get the neighbor and run DFS
                        self.DFS(queue, p+1)

            # if there are some pixel unflood, create a component to flood it
            if not self.with_marker:
                while np.any(self.maze == 0):
                    self.players += 1
                    init_pos = np.argwhere(self.maze == 0)[0]
                    queue = deque(init_pos[np.newaxis,:])
                    self.DFS(queue, self.players)

            # get watershed from maze
            self.watershed = np.where(self.maze == -1, self.watershed, self.maze)
            self.watersheds.append(self.watershed)
            print('water level: ', water_level, end = '\r')

        print('\nFinished watersheding!')
```



```
    def region_merging(self, similarity_threshold, apply = True):
        RAG_copy = self.RAG.copy()
        watershed_copy = np.copy(self.watershed)

        any_smaller = True    # flag for check whether there are similar component
        while any_smaller:
            any_smaller = False
            for e in RAG_copy.edges:
                # use mean as the metric of similarity
                region1 = self.image[watershed_copy == e[0]]
                region2 = self.image[watershed_copy == e[1]]
                rigion_mean1 = np.mean(region1)
                rigion_mean2 = np.mean(region2)
                if abs(rigion_mean2 - rigion_mean1) < similarity_threshold:
                    any_smaller = True

                    # collect merged edge in watershed, after will delete it
                    bound = RAG_copy[e[0]][e[1]]['bound']
                    bound_merge = []

                    # merged the bound of the component
                    commom_neighbors = sorted(nx.common_neighbors(RAG_copy, e[0], e[1]))
                    for n in commom_neighbors:
                        bound1 = RAG_copy[e[0]][n]['bound']
                        bound2 = RAG_copy[e[1]][n]['bound']
                        bound1.extend(bound2)
                        bound_merge.append(bound1)
                    # merge node
                    RAG_copy = nx.contracted_nodes(RAG_copy, e[0],e[1])
                    RAG_copy.remove_edge(e[0], e[0])
                    watershed_copy[watershed_copy == e[1]] = e[0]

                    # merged the bound of the component
                    for i, n in enumerate(commom_neighbors):
                        RAG_copy[e[0]][n]['bound'] = bound_merge[i]

                    # delete the merged edge
                    for b in bound:
                        watershed_copy[b[0],b[1]] = e[0]

        if apply:
            self.RAG = RAG_copy.copy()
            self.watershed = np.copy(watershed_copy)

        return watershed_copy
```

```
    def random_watershed_color_dictionary(self):
        max_val = np.max(self.watershed)
        color_dict = {0: [0,0,0], -2:[255,255,255]}
        for i in range(1,max_val+1):
            color_dict[i] = list(np.random.choice(range(256), size=3))
        return color_dict
```

```
    def show_segmentation(self):
        img = np.where(self.watershed == -2, 255, 0).astype('uint8') # remember to change type
        show_image(img, 'segmentation')

    def save_segmentation(self, dir_pos = ''):
        img = np.where(self.watershed == -2, 255, 0).astype('uint8') # remember to change type
        print('writing to', dir_pos, ':', cv2.imwrite(dir_pos, img))
```

```
    def generate_watershed_gif(self, dir_pos):
        # random color to every connected component
        color_dict = self.random_watershed_color_dictionary()

        watersheds_gif = []    # collect all colored watershed
        watersheds_gif.append(self.image.astype('uint8'))
        for frame in self.watersheds:
            H,W = frame.shape
            frame3D = np.repeat(self.image[:, :, np.newaxis], 3, axis=2)
            for h in range(H):
                for w in range(W):
                    if frame[h,w] != 0:
                        frame3D[h,w,:] = color_dict[frame[h,w]]
            frame3D = frame3D.astype('uint8')
            frame3D = cv2.cvtColor(frame3D, cv2.COLOR_BGR2RGB)
            watersheds_gif.append(frame3D)

        # make collected colored watershed to gif image
        imageio.mimsave(dir_pos, watersheds_gif)
```

```
    def generate_color_watershad(self):
        color_dict = self.random_watershed_color_dictionary()
        self.color_dict_previous = color_dict
        H,W = self.watershed.shape
        output = np.repeat(self.watershed[:, :, np.newaxis], 3, axis=2)
        for h in range(H):
            for w in range(W):
                output[h,w,:] = color_dict[self.watershed[h,w]]
        return output.astype('uint8')

    def show_watershad(self):
        color_watershed = self.generate_color_watershad()
        show_image(color_watershed, 'watershed')

    def save_watershad(self, dir_pos = ''):
        color_watershed = self.generate_color_watershad()
        print('writing to', dir_pos, ':', cv2.imwrite(dir_pos, color_watershed))

    def show_segmentation_on_image(self, watershed = np.array([])):
        if watershed.size == 0:
            watershed = self.watershed
        img = np.where(watershed == -2, 255, self.raw_image).astype('uint8')
        show_image(img, 'segmentation_on_image')

    def save_segmentation_on_image(self, watershed = np.array([]), dir_pos = ''):
        if watershed.size == 0:
            watershed = self.watershed
        img = np.where(watershed == -2, 255, self.raw_image).astype('uint8')
        print('writing to', dir_pos, ':', cv2.imwrite(dir_pos, img))
```

```
    def save_drawing_image(self, dir_pos = ''):
        print('writing to', dir_pos, ':', cv2.imwrite(dir_pos, self.drawing_image.astype('uint8')))

    def save_marker_image(self, dir_pos = ''):
        marker_watershed = self.watersheds[0]
        H,W = marker_watershed.shape
        output = np.repeat(self.raw_image[:, :, np.newaxis], 3, axis=2)
        for h in range(H):
            for w in range(W):
                if marker_watershed[h,w] != 0:
                    output[h,w,:] = self.color_dict_previous[marker_watershed[h,w]]
        print('writing to', dir_pos, ':', cv2.imwrite(dir_pos, output.astype('uint8')))
```