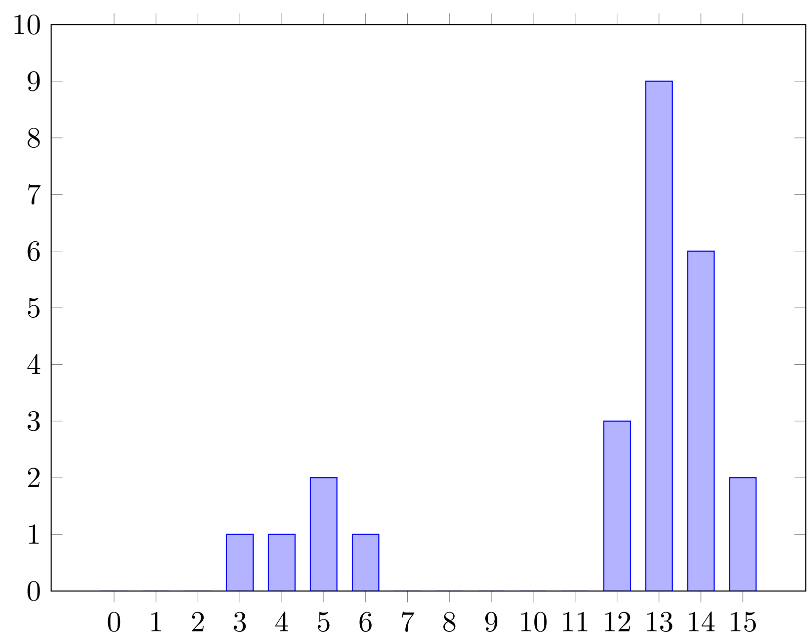
Computer Vision
Assignment 9

Alireza Moradi

November 25, 2020

1 Histogram



1.1

- Mean: 11.6
- Median: 13
- Mod: 13
- Variance: $\sigma^2 = \dfrac{\Sigma(x_i - \mu)^2}{n} = 13.04$

1.2

`Group I`(< 12):

- Mean: 4.6
- Median: 5
- Mod: 5
- Variance: $\sigma^2 = \dfrac{\Sigma(x_i - \mu)^2}{n} = 1.04$

`Group II`(≥ 12):

- Mean: 13.35
- Median: 13
- Mod: 13
- Variance: $\sigma^2 = \dfrac{\Sigma(x_i - \mu)^2}{n} = 0.7275$

1.3

We have 16 colors (0 to 15) here, So we have to compute 16 times ($O(n)$).

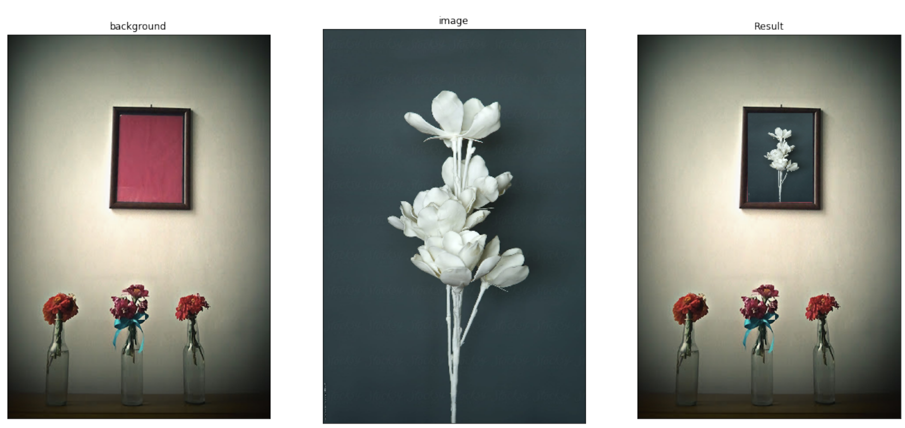
- 1 : `6.355555555555555`
- 2 : `6.596938775510204`
- 3 : `6.840236686390532`
- 4 : `7.047499999999999`
- 5 : `7.287960330578512`
- 6 : `7.388488888888889`
- 7 : `7.702897455278408`
- 8 : `8.096875`
- 9 : `8.420357772738727`
- 10 : `8.534444444444444`
- 11 : `8.08595041322314`
- 12 : `6.081944444444445`
- 13 : `5.863773833004602`
- 14 : `4.988775510204081`
- 15 : `5.994311111111112`

So according to variances regarding each threshold, We can see that when the threshold is `13` we have the lowest variance, Thus this is the best possible threshold.

2 Implementations

2.1 Transformation

First I select 4 corners of flower image. Then read the 4 corners of picture frame from a `csv` file I saved before. Then I use `findHomography` and `warpAffine` to find the transformation matrix and apply it. Finally I `or` the resulting image and the picture frame to get the final result.

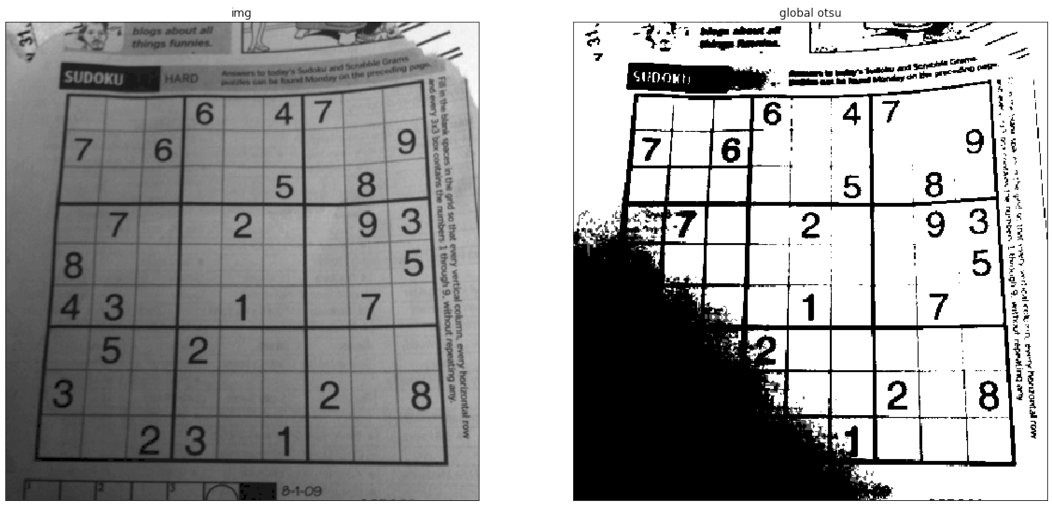


```python
def AR(background, image):
    '''
    Adds the input image to the background image properly.

    Parameters:
        background (numpy.ndarray) : background image
        image (numpy.ndarray): input image

    Returns:
        numpy.ndarray: The result image.
    '''
    print(background.shape)
    print(image.shape)
    result = background.copy()
    h,w,d = image.shape
    print(h,w,d)
    src = np.array([np.array([0, 0]),
                  np.array([ 0,h-1]),
                  np.array([ w-1,h-1,]),
                  np.array([w-1,0])],dtype='float32')

    with open("back.csv") as csv_file:
        import csv
        csv_reader = csv.reader(csv_file, delimiter=",")
        dst = []
        for i, row in enumerate(csv_reader):
            try:
                dst.append(np.array([row[1], row[2]]))
            except ValueError:
                continue
    dst = np.array(dst, dtype='float32')

    print(src)
    print(dst)
    M, _ = cv2.findHomography(src, dst)
    M = M[:2,:]
    print(M.shape)
    transformed_image = cv2.warpAffine(image,M,(result.shape[1], result.shape[0]))

    for i in range(transformed_image.shape[0]):
        for j in range(transformed_image.shape[1]):
            if transformed_image[i][j].all() == 0:
                continue
            else:
                result[i][j] = transformed_image[i][j]

    return result
```

2.2 OTSU

2.2.1 Global

In a loop iterating all possible thresholds, I split the histogram to 2 slices and compute the variance of each slice and add them together weighted by the number of pixels in each slice. Finally the threshold with the lowest total variance is selected and pixels above that threshold become 255 and pixels lower than the threshold become 0.

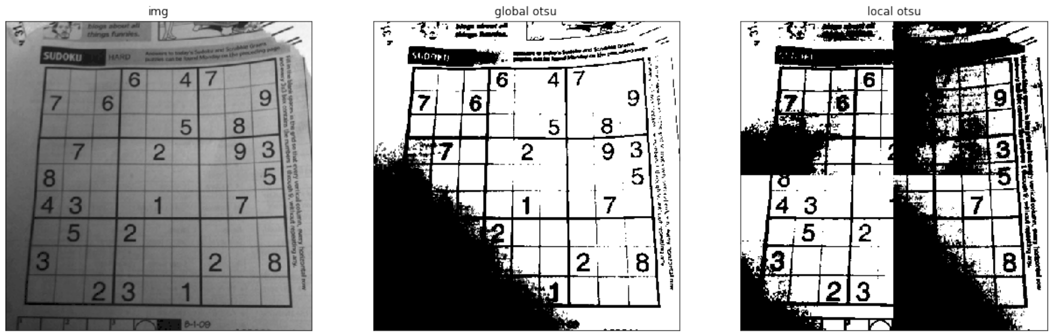



```python
def global_otsu(image):
    '''
    Applys global otsu on the input image.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        numpy.ndarray: The result panorama image.
    '''

    out_img = image.copy()

    #Write your code here
    total_pixels = out_img.shape[0] * out_img.shape[1]
    mean_weigth = 1.0/total_pixels
    hist, bins = np.histogram(out_img, np.array(range(0, 256)))
    final_thresh = -1
    final_value = float('inf')
    for thresh in bins[1:-1]:
        Wb = np.sum(hist[:thresh]) / total_pixels
        Wf = np.sum(hist[thresh:]) / total_pixels

        mub = np.var(hist[:thresh])
        muf = np.var(hist[thresh:])
```

```
        value = Wb * (mub ** 1) + Wf * (muf ** 1)

        if value < final_value:
            final_thresh = thresh
            final_value = value

    out_img[out_img > final_thresh] = 255
    out_img[out_img < final_thresh] = 0

    return out_img
```

2.2.2 Local

In this part I just splitted the image to 4 equal slices and called `global_otsu` for each slice and concatenated the 4 resulting images back together.

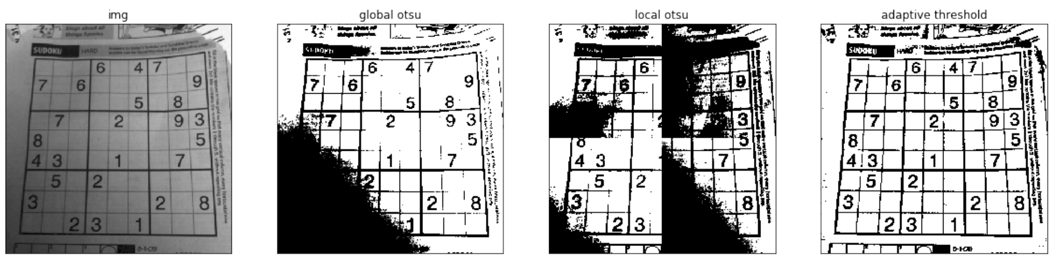



```python
def local_otsu(image):
    '''
    Applys local otsu on the input image.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        numpy.ndarray: The result panorama image.
    '''

    h, w = image.shape
    out_img = image.copy()
    top_left = image[:h//2,:w//2]
    bottom_left = image[h//2:,:w//2]
    top_right = image[:h//2,w//2:]
    bottom_right = image[h//2:,w//2:]

    top_left = global_otsu(image[:h//2,:w//2])
    bottom_left = global_otsu(image[h//2:,:w//2])
    top_right = global_otsu(image[:h//2,w//2:])
    bottom_right = global_otsu(image[h//2:,w//2:])
    image = np.concatenate([np.concatenate([top_left,top_right],axis=-1),
                            np.concatenate([bottom_left, bottom_right],axis=-1)])
    return image
```

2.2.3 Adaptive Threshold

- `maxValue`: Non-zero value assigned to the pixels for which the condition is satisfied.
- `adaptiveMethod`: Adaptive thresholding algorithm to use (Mean or Gaussian). The `BORDER_REPLICATE` | `BORDER_ISOLATED` is used to process boundaries.
- `thresholdType`: Thresholding type that must be either `THRESH_BINARY` or `THRESH_BINARY_INV`.
- `blockSize`: Size of a pixel neighborhood that is used to calculate a threshold value for the pixel: 3, 5, 7, and so on.
- `C`: Constant subtracted from the mean or weighted mean. Normally, it is positive but may be zero or negative as well.

As it can be seen in the images, `adaptiveThreshold` has a better output because it uses a small window around each pixel to apply the procedure (like `CLAHE`). In `OTSU` because we use all of the image to detect the thresholds and we have a shadow in the bottom left side of the image, some of the data will be lost.

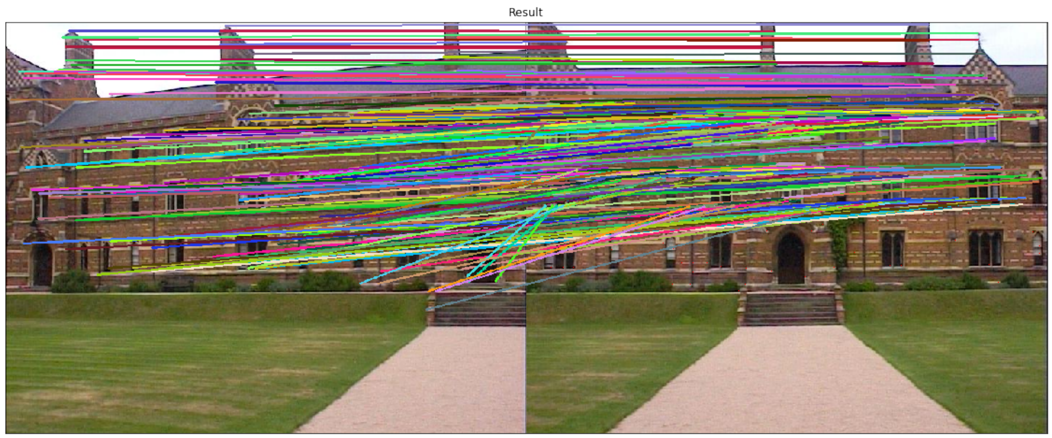


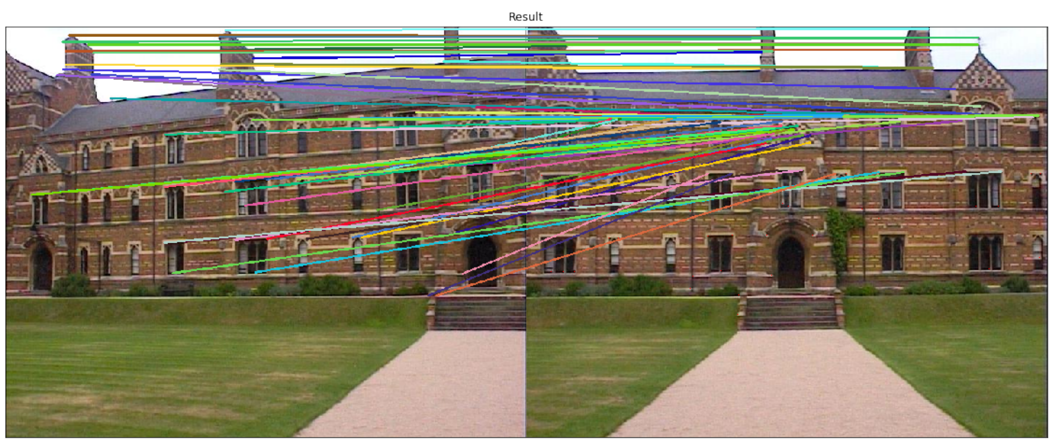

```python
def adaptive_th(image):
    '''
    Applys adaptive threshold on the input image.

    Parameters:
        image (numpy.ndarray): The input image.

    Returns:
        numpy.ndarray: The result panorama image.
    '''

    out_img = image.copy()

    #Write your code here
    out_img = cv2.adaptiveThreshold(src=out_img,
                                    maxValue=255,
                                    adaptiveMethod=1,
                                    thresholdType=cv2.THRESH_BINARY,
                                    blockSize=75,
                                    C=5)

    return out_img
```

2.3 Keypoint Matching

First I computed keypoints using `Harris`. Then I removed the points which were to close to each other and those that would go out of border of the image (based on the `window_size`)). At last I computed `NCC` for all of the keypoints and match those that were highly correlated. As you can see in the resulting image, `NCC` is not a good choice for matching keypoints.

`NCC` is neither rotation nor scale invariant. NCC has a very simple descriptor and does not benefit from stability enhancements built into something like `SIFT` such as invariance, distinctiveness from high dimension and the elimination of detected points at low contrast and or low edge response regions.
The 2 images below just have different thresholds for detecting keypoints.






```python
def correlation_coefficient(patch1, patch2):
    return cv2.matchTemplate(patch1,patch1,cv2.TM_CCORR_NORMED)

def euclidean_distance(point1, point2):
    x1,y1 = point1
    x2,y2 = point2
    return np.sqrt(np.square(x1-x2) + np.square(y1-y2))
```

```python
def find_match(image1, image2):
    '''
    Finds match points between two input images.

    Parameters:
        image1 (numpy.ndarray): input image.
        image2 (numpy.ndarray): second input image.

    Returns:
        numpy.ndarray: The result image.
    '''

    image1_cop = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY)
    image2_cop = cv2.cvtColor(image2, cv2.COLOR_BGR2GRAY)
    image1_cop = np.float32(image1_cop)
    image2_cop = np.float32(image2_cop)
    t = 250

    harris1 = cv2.cornerHarris(image1_cop,2,5,k=0.07)
    harris2 = cv2.cornerHarris(image2_cop,2,5,k=0.07)
    dest1 = cv2.dilate(harris1, None)
    dest2 = cv2.dilate(harris2, None)
    keypoints1 = dest1 > 0.02 * dest1.max()
    keypoints2 = dest2 > 0.02 * dest2.max()
    img1 = image1.copy()
    img2 = image2.copy()

    window_size = 9
    r = window_size//2

    k1 = np.argwhere(keypoints1)
    k2 = np.argwhere(keypoints2)
    euc_thresh = 3

    print(k1.shape, k2.shape)

    for i in range(len(k1)-1):
        for j in range(i+1,len(k1)):
            if euclidean_distance(k1[i],k1[j]) < euc_thresh:
                x,y = k1[j]
                keypoints1[x][y] = False

    for i in range(len(k2)-1):
        for j in range(i+1,len(k2)):
            if euclidean_distance(k2[i],k2[j]) < euc_thresh:
                x,y = k2[j]
                keypoints2[x][y] = False

```

```python
    for i in range(len(k1)):
        h ,w = keypoints1.shape
        x ,y = k1[i]
        if x - r < 0 or x + r >= h or y - r < 0 or y + r >= w:
            keypoints1[x][y] = False

    for i in range(len(k2)):
        h ,w = keypoints2.shape
        x ,y = k2[i]
        if x - r < 0 or x + r >= h or y - r < 0 or y + r >= w:
            keypoints2[x][y] = False

    k1 = np.argwhere(keypoints1)
    k2 = np.argwhere(keypoints2)

    img1[keypoints1] = [255,0,0]
    img2[keypoints2] = [255,0,0]

    result = np.concatenate([img1,img2],axis=-2)

    print(k1.shape, k2.shape)

    for x1,y1 in k1:
        best_match = None
        best_idx = None
        best_value = float('-inf')
        for idx,(x2,y2) in enumerate(k2):
            corr = correlation_coefficient(image1[x1-r:x1+r+1,y1-r:y1+r+1],
                                            image2[x2-r:x2+r+1,y2-r:y2+r+1])
            if corr > best_value:
                best_value = corr
                best_idx = idx
                best_match = (y2 + keypoints2.shape[1],x2)
        if best_idx is None:
            continue
        k2 = np.delete(k2,(best_idx),axis=0)
        cv2.line(result, (y1,x1), best_match, (np.random.randint(0,256),
                                               np.random.randint(0,256),
                                               np.random.randint(0,256)), 2)

    return result
```