Computer Vision
Assignment 7

Alireza Moradi

November 7, 2020

1 Color Spaces

a.

- $K = 1 - max(R, G, B) = 1 - \frac{max(0,255,100)}{255} = 0$
- $C = 1 - R - K = (1 - \frac{0}{255} - 0) \times 255 = 255$
- $M = 1 - G - K = (1 - \frac{255}{255} - 0) \times 255 = 0$
- $Y = 1 - B - K = (1 - \frac{100}{255} - 0) \times 255 = 155$

$$\begin{bmatrix} 0 \\ 255 \\ 100 \end{bmatrix} \xrightarrow{RGB\ to\ CMYK} \begin{bmatrix} 255 \\ 0 \\ 155 \\ 0 \end{bmatrix}$$

b.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \Rightarrow \begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} \frac{80}{255} \\ \frac{43}{255} \\ \frac{100}{255} \end{bmatrix} \xrightarrow{CMY\ to\ RGB} \begin{bmatrix} 175 \\ 212 \\ 155 \end{bmatrix}$$

c.

- $R = 255 \times (1 - C) \times (1 - K) = 255 \times (1 - \frac{115}{255}) \times (1 - \frac{155}{255}) \approx 55$
- $G = 255 \times (1 - M) \times (1 - K) = 255 \times (1 - \frac{87}{255}) \times (1 - \frac{155}{255}) \approx 66$
- $B = 255 \times (1 - Y) \times (1 - K) = 255 \times (1 - \frac{0}{255}) \times (1 - \frac{155}{255}) = 100$

$$\begin{bmatrix} 115 \\ 87 \\ 0 \\ 155 \end{bmatrix} \xrightarrow{CMYK\ to\ RGB} \begin{bmatrix} 55 \\ 66 \\ 100 \end{bmatrix}$$

2 Normalized Cross-Correlation

- **Cross-Correlation**: To detect a level of correlation between two signals we use cross-correlation. It is calculated simply by multiplying and summing two-time series together. Using the cross-correlation formula below we can calculate the level of correlation between series.

$$corr(x, y) = \sum_{n=0}^{n-1} x[n] * y[n]$$

- **Normalized Cross-Correlation**: There are three problems with cross-correlation:

  1. It is difficult to understand the scoring value.
  2. Both metrics must have the same amplitude. If Graph B has the same shape as Graph A but values two times smaller, the correlation will not be detected.
  3. Due to the formula, a zero value will not be taken into account.

  To solve these problems we use normalized cross-correlation:

$$norm\_corr(x, y) = \frac{\sum_{n=0}^{n-1} x[n] * y[n]}{\sqrt{\sum_{n=0}^{n-1} x[n]^2 * \sum_{n=0}^{n-1} y[n]^2}}$$

  - Normalized cross-correlation scoring is easy to understand:
    - The higher the value, the higher the correlation is.
    - The maximum value is 1 when two signals are exactly the same:

$$norm_corr(a, a) = 1$$

    - The minimum value is -1 when two signals are exactly opposite:

$$norm_corr(a, -a) = -1$$

  - Normalized cross-correlation can detect the correlation of two signals with different amplitudes:

$$norma_corr(a, a/2) = 1$$

    Notice we have perfect correlation between signal A and the same signal with half the amplitude!

  The normalized cross-correlation (NCC), usually its 2D version, is routinely encountered in template matching algorithms, such as in facial recognition, motion-tracking, registration in medical imaging, etc. Its rapid computation becomes critical in time sensitive applications.

3 Hessian Detector

The Hessian affine detector algorithm is almost identical to the Harris affine region detector. The Harris affine detector relies on interest points detected at multiple scales using the Harris corner

measure on the second-moment matrix. The Hessian affine also uses a multiple scale iterative algorithm to spatially localize and select scale and affine invariant points. However, at each individual scale, the Hessian affine detector chooses interest points based on the Hessian matrix at that point:

$$H(x) = \begin{bmatrix} L_{xx}(x) & L_{xy}(x) \\ L_{yx}(x) & L_{yy}(x) \end{bmatrix}$$

where $L_{aa}(\mathbf{x})$ is second partial derivative in the a direction and $L_{ab}(\mathbf{x})$ is the mixed partial second derivative in the a and b directions. It's important to note that the derivatives are computed in the current iteration scale and thus are derivatives of an image smoothed by a Gaussian kernel: $L(\mathbf{x}) = g(\sigma_I) \otimes I(\mathbf{x})$. As discussed in the Harris affine region detector article, the derivatives must be scaled appropriately by a factor related to the Gaussian kernel: σ_I^2.

At each scale, interest points are those points that simultaneously are local extrema of both the determinant and trace of the Hessian matrix. The trace of Hessian matrix is identical to the Laplacian of Gaussians (LoG):

$$DET = \sigma_I^2(L_{xx}L_{yy}(\mathbf{x}) - L_{xy}^2(\mathbf{x}))$$
$$TR = \sigma_I(L_{xx} + L_{yy})$$

By choosing points that maximize the determinant of the Hessian, this measure penalizes longer structures that have small second derivatives (signal changes) in a single direction. Like the Harris affine algorithm, these interest points based on the Hessian matrix are also spatially localized using an iterative search based on the Laplacian of Gaussians. Predictably, these interest points are called Hessian–Laplace interest points. Furthermore, using these initially detected points, the Hessian affine detector uses an iterative shape adaptation algorithm to compute the local affine transformation for each interest point.

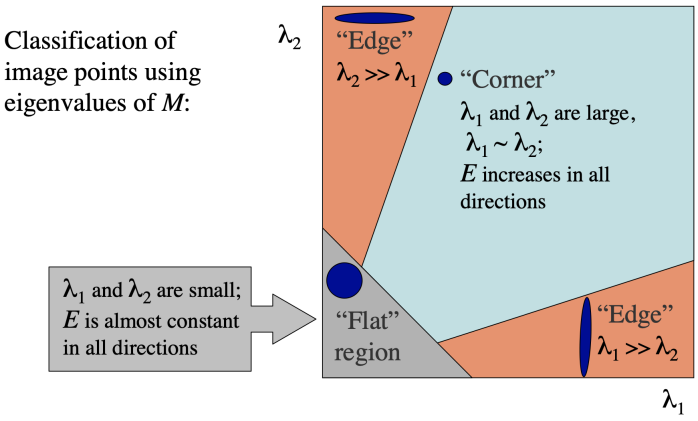
The implementation of this algorithm is almost identical to that of the Harris affine detector; however, the above mentioned Hessian measure replaces all instances of the Harris corner measure.

4 Eigen Values

```
m1 = np.array([[84.33,-16.97],[-16.97,59.48]])
print("M1: ",np.round(np.linalg.eig(m1)[0],3))

m2 = np.array([[163.54,-0.217],[-0.217,0.1053]])
print("M2: ",np.round(np.linalg.eig(m2)[0],3))

m3 = np.array([[0.1714,-0.496],[-0.496,164.4]])
print("M3: ",np.round(np.linalg.eig(m3)[0],3))

m4 = np.array([[0.1439,-0.009],[-0.009,0.323]])
print("M4: ",np.round(np.linalg.eig(m4)[0],3))
```

```
M1:  [92.937 50.873]
M2:  [1.6354e+02 1.0500e-01]
M3:  [  0.17  164.401]
M4:  [0.143 0.323]
```

- $M_1 : \begin{pmatrix}\lambda_1 & \lambda_2\end{pmatrix} = \begin{pmatrix}92.937 & 50.873\end{pmatrix}$
- $M_2 : \begin{pmatrix}\lambda_1 & \lambda_2\end{pmatrix} = \begin{pmatrix}163.54 & 0.105\end{pmatrix}$
- $M_3 : \begin{pmatrix}\lambda_1 & \lambda_2\end{pmatrix} = \begin{pmatrix}0.17 & 164.401\end{pmatrix}$
- $M_4 : \begin{pmatrix}\lambda_1 & \lambda_2\end{pmatrix} = \begin{pmatrix}0.143 & 0.323\end{pmatrix}$



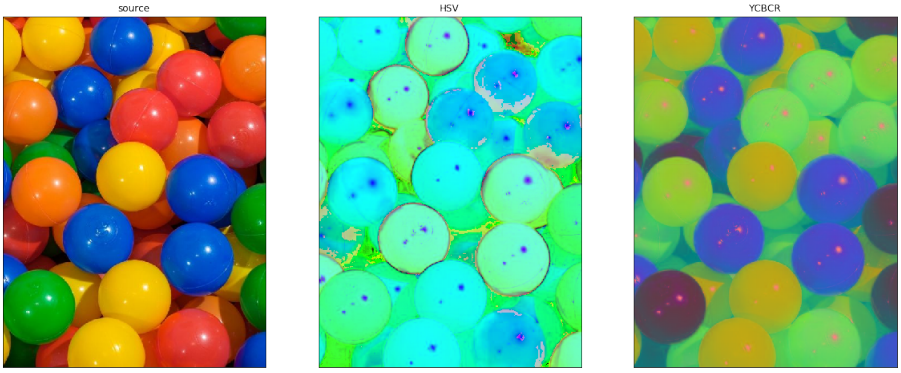
1. When λ_1 and λ_2 are small, the region is **flat**.
2. When $\lambda_1 \gg \lambda_2$ or vice versa, the region is an **edge**.
3. When λ_1 and λ_2 are large and $\lambda_1 \approx \lambda_2$, the region is a **corner**.

So according to above explanation, M_1 is a corner (**green** circle), M_2 and M_3 are edges (**blue** and **red** circles respectively) and M_4 is a flat region (**yellow** circle).

5 Implementations

5.1 Color Spaces

5.1.1 Changing Color Spaces

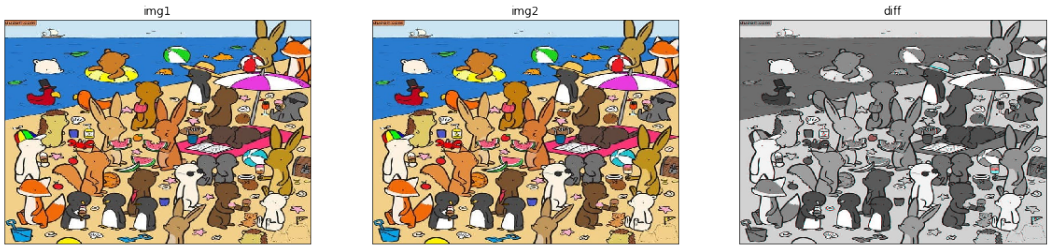


```python
def convert_to_hsv(image):
    '''
    Converts the color space of the input image to the HSV color space.

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

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

    out_img = image.copy()

    #Write your code here
    out_img = cv2.cvtColor(out_img,cv2.COLOR_BGR2HSV)

    return out_img
```

```python
def convert_to_ycbcr(image):
    '''
    Converts the color space of the input image to the YCbCr color space.

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

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

    out_img = image.copy()

    #Write your code here
    out_img = cv2.cvtColor(out_img,cv2.COLOR_BGR2YCR_CB)

    return out_img
```

5.1.2 Detecting Differences

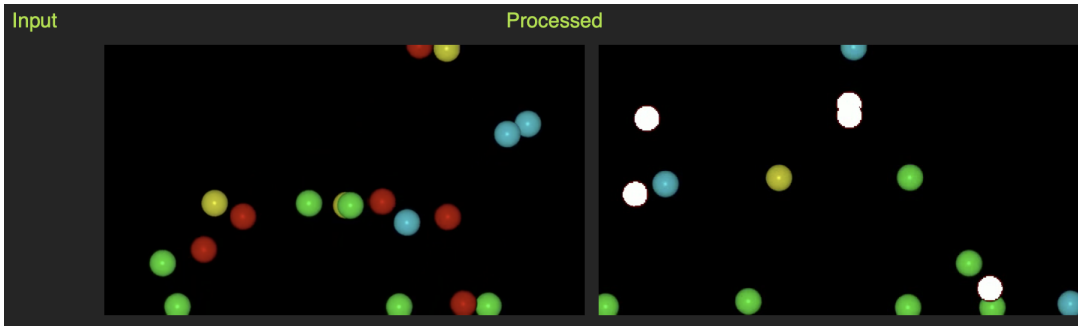
I did exactly what was said in the class. I put one of the images in the `R` and the other image in `G` and `B`, So where ever we have a difference in these images, Because the `RGB` channels will have different values, That area will be highlighted.




```python
def get_dif(image1, image2):
    '''
    Creates a new image that differences between two input images are shown.

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

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

    out_img1 = image1.copy()
    out_img2 = image2.copy()
    img_rgb = cv2.merge((out_img1,out_img2,out_img2))
    return img_rgb
```

5.2 Change Color of Circles

I used `opencv.inRange` function which identifies areas with colors inside a certain range, And changed the color of these areas to white.
Source



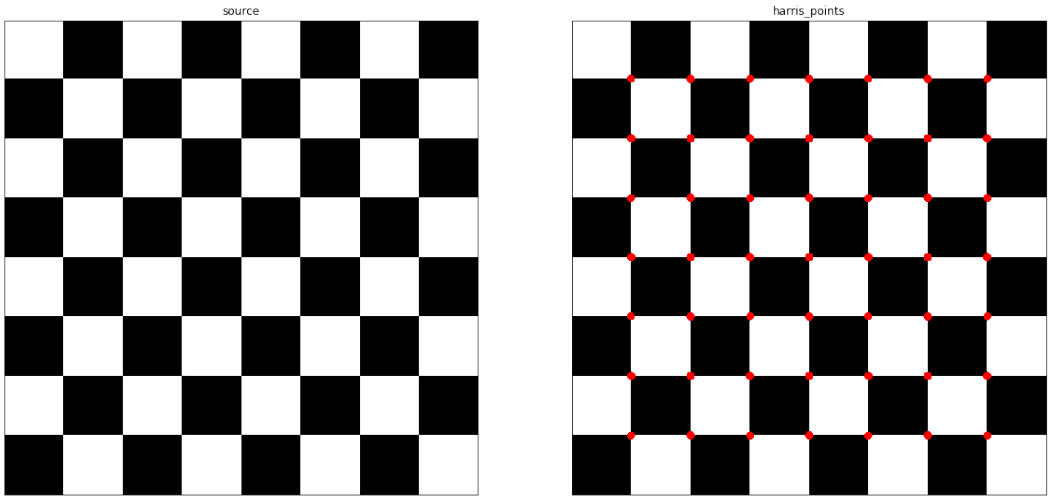

```python
def process_frame(frame):
    '''
    Converts red circles in the input image to white circles.

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

    Returns:
        numpy.ndarray: The result output frame.
    '''

    result = frame.copy()

    #Write your code here
    low_bound = np.array([0,0,100],dtype=np.uint8)
    high_bound = np.array([30,30,255], dtype=np.uint8)
    result = cv2.inRange(result, low_bound, high_bound)
    mask = np.where(result != 0)
    frame[mask] = [255,255,255]

    result = cv2.cvtColor(result, cv2.COLOR_GRAY2BGR)

    return frame
```

5.3 Harris Corner Detector

First I converted the image to grayscale. Then computed the image gradient in 2 direction using `np.gradient`. Finally I applied a Gaussian Blur on the gradients and computed $R = \det(M) - k \cdot \mathrm{tr}(M)^2$ where M is:

$$M = \sum_{(x,y)\in W} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \begin{bmatrix} \sum_{(x,y)\in W} I_x^2 & \sum_{(x,y)\in W} I_x I_y \\ \sum_{(x,y)\in W} I_x I_y & \sum_{(x,y)\in W} I_y^2 \end{bmatrix}$$

And normalized the result and applied a threshold on it in order to find the corners.
Source




```python
def harris_points(image):
    '''
    Gets corner points by applying the harris detection algorithm.

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

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

    out_img = image.copy()
    window_size = 3
    k = 0.04
    #Write your code here
    out_img = np.float32(cv2.cvtColor(out_img, cv2.COLOR_RGB2GRAY))
    Ix, Iy = np.array(np.gradient(out_img))
    threshold = 0.9

    Ixy = Ix*Iy
    Ix2 = np.square(Ix)
    Iy2 = np.square(Iy)
```

```
23
24      Ixy = cv2.GaussianBlur(Ixy,(5,5),0)
25      Ix2 = cv2.GaussianBlur(Ix2,(5,5),0)
26      Iy2 = cv2.GaussianBlur(Iy2,(5,5),0)
27
28      harris = Ix2*Iy2 - np.square(Ixy) - k * np.square(Ix2 + Iy2)
29
30      cv2.normalize(harris, harris, 0, 1, cv2.NORM_MINMAX)
31
32      loc = np.transpose(np.where(harris >= threshold))
33
34      for i in range(loc.shape[0]):
35          x,y = loc[i]
36          cv2.circle(image, (x,y), 10, (255,0,0), -1)
37
38      return image
```