Image processing

Topics Covered

1. Edge detection
2. Corner Detection
3. Color Detection
4. Face Detection
5. Summary

Edge detection

Edge detection is one of the fundamental operations when we perform image processing. It helps us reduce the amount of data (pixels) to process and maintains the structural aspect of the image. We're going to look into two commonly used edge detection schemes –

a. The gradient (Sobel - first order derivatives) based edge detector and
b. The Laplacian (2nd order derivative, so it is extremely sensitive to noise) based edge detector.

Both of them work with convolutions and achieve the same end goal - Edge Detection.

Sobel Edge Detection

Sobel edge detector is a gradient based method based on the first order derivatives. It calculates the first derivatives of the image separately for the X and Y axes.

The operator uses two 3X3 kernels which are convolved with the original image to calculate approximations of the derivatives - one for horizontal changes, and one for vertical. The picture below shows Sobel Kernels in x-dir and y-dir:

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Laplacian Edge Detection

Unlike the Sobel edge detector, the Laplacian edge detector uses only one kernel. It calculates second order derivatives in a single pass.

A kernel used in this Laplacian detection looks like this:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

If we want to consider the diagonals, we can use the kernel below:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

```
cv2.Laplacian(src, ddepth, other_options...)
```

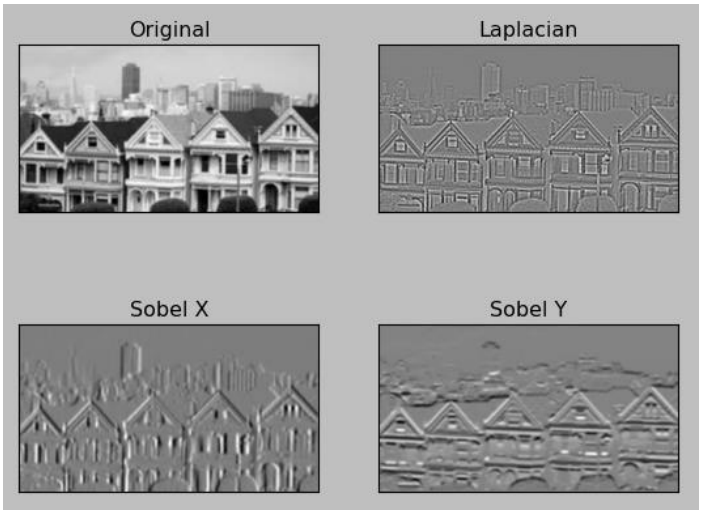
where **ddepth** is the desired depth of the destination image.

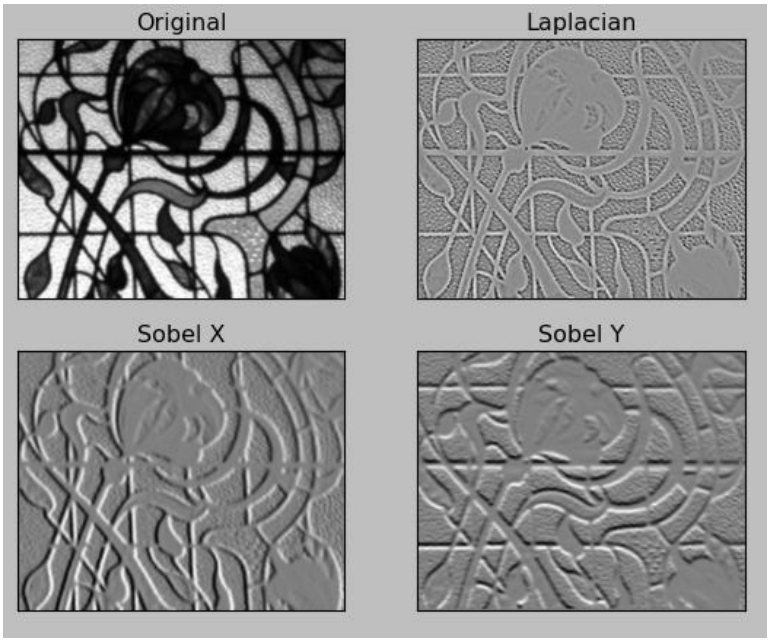
Code for Edge Detection

Here is a code that can do edge detection:

```
import cv2
import numpy as np
from matplotlib import pyplot as plt

# loading image
#img0 = cv2.imread('SanFrancisco.jpg',)
img0 = cv2.imread('windows.jpg',)

# converting to gray scale
gray = cv2.cvtColor(img0, cv2.COLOR_BGR2GRAY)
```

```
# remove noise
img = cv2.GaussianBlur(gray,(3,3),0)

# convolute with proper kernels
laplacian = cv2.Laplacian(img,cv2.CV_64F)
sobelx = cv2.Sobel(img,cv2.CV_64F,1,0,ksize=5)  # x
sobely = cv2.Sobel(img,cv2.CV_64F,0,1,ksize=5)  # y

plt.subplot(2,2,1),plt.imshow(img,cmap = 'gray')
plt.title('Original'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,2),plt.imshow(laplacian,cmap = 'gray')
plt.title('Laplacian'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,3),plt.imshow(sobelx,cmap = 'gray')
plt.title('Sobel X'), plt.xticks([]), plt.yticks([])
plt.subplot(2,2,4),plt.imshow(sobely,cmap = 'gray')
plt.title('Sobel Y'), plt.xticks([]), plt.yticks([])

plt.show()
```


Original
Laplacian
Sobel X
Sobel Y


Original
Laplacian
Sobel X
Sobel Y

Corner Detection

Corners are regions in the image with large variation in intensity in all the directions. One early attempt to find these corners was done by Chris Harris & Mike Stephens in their paper A Combined Corner and Edge Detector in 1988, so now it is called the Harris Corner Detector. He took this simple idea to a mathematical form. It basically finds the difference in intensity for a displacement of (u,v) in all directions. This is expressed as below:

$$E(u,v) = \sum_{x,y} \underbrace{w(x,y)}_{\text{window function}} \underbrace{[I(x+u,y+v)}_{\text{shifted intensity}} - \underbrace{I(x,y)}_{\text{intensity}}]^2$$

The window function is either a rectangular window or a Gaussian window which gives weights to pixels underneath.

We have to maximize this function E(u,v) for corner detection. That means we have to maximize the second term. Applying Taylor Expansion to the above equation and using some mathematical steps (please refer to any standard text books you like for full derivation), we get the final equation as:

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

where

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

Here, Ix and Iy are image derivatives in x and y directions respectively. (These can be easily found using **`cv.Sobel()`**.

Then comes the main part. After this, they created a score, basically an equation, which determines if a window can contain a corner or not.

$$R = \det(M) - k(\operatorname{trace}(M))^2$$

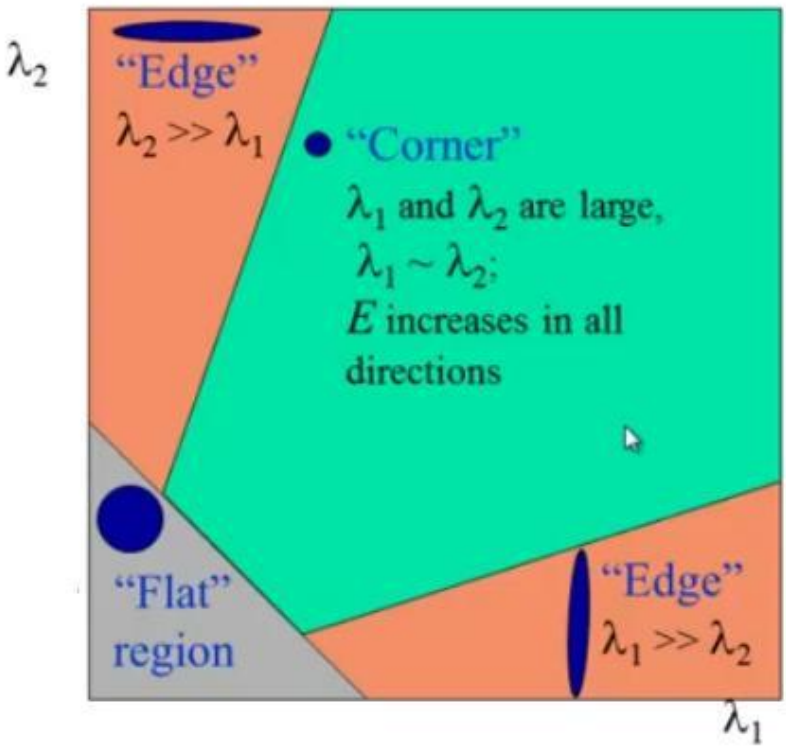
where

- det(M)=λ1λ2
- trace(M)=λ1+λ2
- λ1 and λ2 are the eigenvalues of M

So the magnitudes of these eigenvalues decide whether a region is a corner, an edge, or flat.

- When |R| is small, which happens when λ1 and λ2 are small, the region is flat.
- When R<0, which happens when λ1>>λ2 or vice versa, the region is edge.
- When R is large, which happens when λ1 and λ2 are large and λ1∼λ2, the region is a corner.

It can be represented in a picture as follows:



So the result of Harris Corner Detection is a grayscale image with these scores. Thresholding for a suitable score gives you the corners in the image. We will do it with a simple image.

Harris Corner Detector in OpenCV

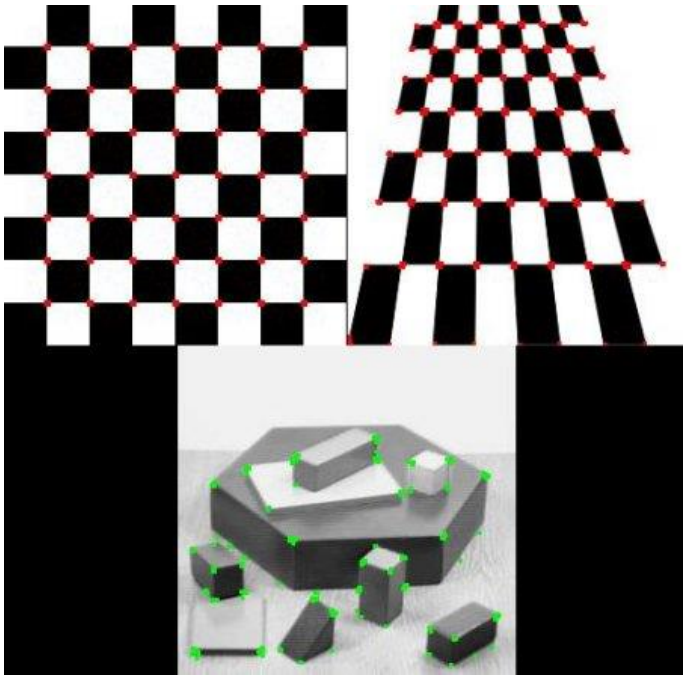
OpenCV has the function **`cv.cornerHarris()`** for this purpose. Its arguments are:

- **img** - Input image. It should be grayscale and float32 type.
- **blockSize** - It is the size of neighbourhood considered for corner detection
- **ksize** - Aperture parameter of the Sobel derivative used.
- **k** - Harris detector free parameter in the equation.

See the example below:

```
import numpy as np
import cv2 as cv
filename = 'chessboard.png'
img = cv.imread(filename)
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
gray = np.float32(gray)
dst = cv.cornerHarris(gray,2,3,0.04)

#result is dilated for marking the corners, not important
dst = cv.dilate(dst,None)

# Threshold for an optimal value, it may vary depending on the image.
img[dst>0.01*dst.max()]=[0,0,255]
cv.imshow('dst',img)
if cv.waitKey(0) & 0xff == 27:
cv.destroyAllWindows()
```

Below are the three results:



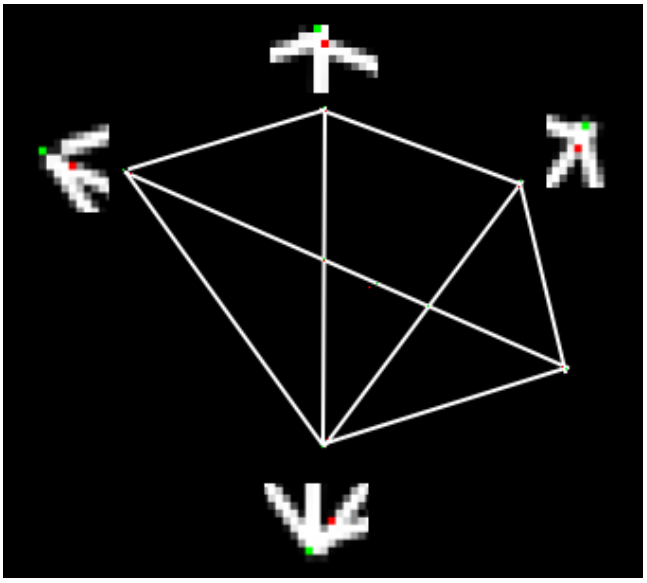
Corner with SubPixel Accuracy

Sometimes, you may need to find the corners with maximum accuracy. OpenCV comes with a function **`cv.cornerSubPix()`** which further refines the corners detected with sub-pixel accuracy. Below is an example. As usual, we need to find the Harris corners first. Then we pass the centroids of these corners (There may be a bunch of pixels at a corner, we take their centroid) to refine them. Harris corners are marked in red pixels and refined corners are marked in green pixels. For this function, we have to define the criteria when to stop the iteration. We stop it after a specified number of iterations or a certain accuracy is achieved, whichever occurs first. We also need to define the size of the neighbourhood it searches for corners.

```
import numpy as np
import cv2 as cv
filename = 'chessboard2.jpg'
img = cv.imread(filename)
gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)

# find Harris corners
gray = np.float32(gray)
dst = cv.cornerHarris(gray,2,3,0.04)
dst = cv.dilate(dst,None)
ret, dst = cv.threshold(dst,0.01*dst.max(),255,0)
dst = np.uint8(dst)
```

```
# find centroids
ret, labels, stats, centroids = cv.connectedComponentsWithStats(dst)

# define the criteria to stop and refine the corners
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 100, 0.001)
corners = cv.cornerSubPix(gray,np.float32(centroids),(5,5),(-1,-1),criteria)

# Now draw them
res = np.hstack((centroids,corners))
res = np.int0(res)
img[res[:,1],res[:,0]]=[0,0,255]
img[res[:,3],res[:,2]] = [0,255,0]
cv.imwrite('subpixel5.png',img)
```

Below is the result, where some important locations are shown in the zoomed window to visualize:



Color Detection

Color detection is the process of detecting the name of any color. Well, for humans this is an extremely easy task but for computers, it is not straightforward. Human eyes and brains work together to translate light into color. Light receptors that are present in our eyes transmit the signal to the brain. Our brain then recognizes the color. Since childhood, we have mapped certain lights with their color names. We will be using the somewhat same strategy to detect color names.

Colors are made up of 3 primary colors; red, green, and blue. In computers, we define each color value within a range of 0 to 255. So in how many ways we can define a color? The answer is 256*256*256 = 16,581,375. There are approximately 16.5 million different ways to represent a color.

The color detection process is mostly in demand in computer vision. A color detection algorithm identifies pixels in an image that match a specified color or color range. The color of detected pixels can then be changed to distinguish them from the rest of the image. This process can be easily done using OpenCV.

In this article, we will import two modules - cv2 and numpy. After this, we will load the image using imread, we will convert the color-space from BGR to HSV using `cv2.cvtColor()`, like the following –

```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
```

Next, we will define the lower and upper color(blue) rgb value in two variables, i.e.-

```
light_blue = np.array([110,50,50])
dark_blue = np.array([130,255,255])
```

The **OpenCV** provides **cv2.inRange()** method to set the color range. It accepts three parameters, the source image in the first parameter and lower and upper color boundary of the threshold region. This function returns a binary mask, which we will pass in the bitwise AND operator.

```
mask = cv2.inRange(hsv, light_blue, dark_blue)

output = cv2.bitwise_and(image,image, mask= mask)
```

Face Detection

Face detection is a computer vision problem that involves finding faces in photos. It is a trivial problem for humans to solve and has been solved reasonably well by classical feature-based techniques, such as the cascade classifier. More recently deep learning methods have achieved state-of-the-art results on standard benchmark face detection datasets. One example is the Multi-task Cascade Convolutional Neural Network, or MTCNN for short.

Locating a face in a photograph refers to finding the coordinate of the face in the image, whereas localization refers to demarcating the extent of the face, often via a bounding box around the face.

Face Detection with OpenCV

Feature-based face detection algorithms are fast and effective and have been used successfully for decades.

Perhaps the most successful example is a technique called cascade classifiers first described by Paul Viola and Michael Jones and their 2001 paper titled “Rapid Object Detection using a Boosted Cascade of Simple Features.”

In the paper, effective features are learned using the AdaBoost algorithm, although importantly, multiple models are organized into a hierarchy or *“cascade”*

In the paper, the AdaBoost model is used to learn a range of very simple or weak features in each face, that together provide a robust classifier.

> feature selection is achieved through a simple modification of the AdaBoost procedure: the weak learner is constrained so that each weak classifier returned can depend on only a single feature. As a result, each stage of the boosting process, which selects a new weak classifier, can be viewed as a feature selection process.

The models are then organized into a hierarchy of increasing complexity, called a "*cascade*". Simpler classifiers operate on candidate face regions directly, acting like a coarse filter, whereas complex classifiers operate only on those candidate regions that show the most promise as faces.

The result is a very fast and effective face detection algorithm that has been the basis for face detection in consumer products, such as cameras.

Their detector, called detector cascade, consists of a sequence of simple-to-complex face classifiers

> A method for combining successively more complex classifiers in a cascade structure which dramatically increases the speed of the detector by focusing attention on promising regions of the image.

and has attracted extensive research efforts. Moreover, detector cascade has been deployed in many commercial products such as smartphones and digital cameras.

It is a modestly complex classifier that has also been tweaked and refined over the last nearly 20 years.

A modern implementation of the Classifier Cascade face detection algorithm is provided in the OpenCV library. This is a C++ computer vision library that provides a python interface. The benefit of this implementation is that it provides pre-trained face detection models, and provides an interface to train a model on your own dataset.

OpenCV provides the CascadeClassifier class that can be used to create a cascade classifier for face detection. The constructor can take a filename as an argument that specifies the XML file for a pre-trained model.

OpenCV provides a number of pre-trained models as part of the installation. These are available on your system and are also available on the OpenCV GitHub project.

Once downloaded, we can load the model as follows:

```
#import required library

import cv2

from cv2 import imread
```

```
from cv2 import imshow

from cv2 import waitKey

from cv2 import destroyAllWindows

from cv2 import CascadeClassifier

from cv2 import rectangle



# load the pre-trained model
classifier = CascadeClassifier('haarcascade_frontalface_default.xml')
```

Once loaded, the model can be used to perform face detection on a photograph by calling the detectMultiScale() function. This function will return a list of bounding boxes for all faces detected in the photograph. We can demonstrate this with an example with the *test.jpg*

```
# load the photograph
pixels = imread('test.jpg')

# perform face detection
bboxes = classifier.detectMultiScale(pixels)

# print bounding box for each detected face
for box in bboxes:
	print(box)
```

The photo can be loaded using OpenCV via the *imread()* function.

The complete example of performing face detection on the photograph with a pre-trained cascade classifier in OpenCV is listed above.

Running the example first loads the photograph, then loads and configures the cascade classifier; faces are detected and each bounding box is printed.

```
[172  77  65  65]
[368  43  74  74]
[276  66  65  65]
```

Each box lists the *x* and *y* coordinates for the bottom-left-hand-corner of the bounding box, as well as the width and the height. The results suggest that two bounding boxes were detected.

We can update the example to plot the photograph and draw each bounding box.

This can be achieved by drawing a rectangle for each box directly over the pixels of the loaded image using the *rectangle()* function that takes two points.
We can then plot the photograph and keep the window open until we press a key to close it.

The complete example is listed below.

Running the example, we can see that the photograph was plotted correctly and that each face was correctly detected.

```
# plot photo with detected faces using opencv cascade classifier
from cv2 import imread
from cv2 import imshow
from cv2 import waitKey
from cv2 import destroyAllWindows
from cv2 import CascadeClassifier
from cv2 import rectangle
# load the photograph
pixels = imread('test.jpg')
# load the pre-trained model
classifier = CascadeClassifier('haarcascade_frontalface_default.xml')
# perform face detection
bboxes = classifier.detectMultiScale(pixels)
# print bounding box for each detected face
for box in bboxes:
# extract
    x, y, width, height = box
    x2, y2 = x + width, y + height
# draw a rectangle over the pixels
    rectangle(pixels, (x, y), (x2, y2), (0,0,255), 1)
# show the image
imshow('face detection', pixels)
# keep the window open until we press a key
waitKey(0)
# close the window
destroyAllWindows()
```

Running the example, we can see that the photograph was plotted correctly and that each face was correctly detected.

Summary

- Edge detection helps us reduce the amount of data (pixels) to process and maintains the structural aspect of the image.
- Sobel edge detector is a gradient based method based on the first order derivatives. It calculates the first derivatives of the image separately for the X and Y axes.
- The Laplacian edge detector uses only one kernel. It calculates second order derivatives in a single pass.
- Corners are regions in the image with large variation in intensity in all the directions.
- A color detection algorithm identifies pixels in an image that match a specified color or color range. The color of detected pixels can then be changed to distinguish them from the rest of the image.
- Face detection is a computer vision problem that involves finding faces in photos. It is a trivial problem for humans to solve and has been solved reasonably well by classical feature-based techniques, such as the cascade classifier.