COMPUTER VISION Detailed Introduction

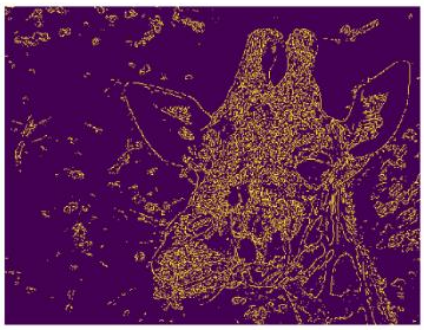
1. Understanding color models and drawing figures on images
2. ***The basics of image processing with filtering***
3. From feature detection to face detection
4. Contour detection and having a little bit of fun

Today we're going to talk about how to manipulate images. These are going to be a preprocessing stage. When it comes to detecting edges and contours, noise gives a great impact on the accuracy of detection. Therefore removing noises and controlling the intensity of the pixel values can help the model to focus on the general details and get higher accuracy. Blurring, thresholding, and morphological transformation are the techniques we use for this purpose. **Source**: https://towardsdatascience.com/computer-vision-for-beginners Jiwon Jeong

Blurring

The goal of blurring is to perform noise reduction. But we have to pay extra care here. If we apply edge detection algorithms to the images with high resolution, we'll get too many detected outcomes that we aren't interested in.

Original Image



Edges without Blur

Edges with Blur

On the contrary, if we blur the images too much, we'll lose the data. Therefore we need to find an adequate amount of blurring we're going to apply without losing desirable edges.

There are several techniques used to achieve blurring effects but we're going to talk about the four major ones used in OpenCV: ***Averaging blurring, Gaussian blurring, median blurring*** *and* ***bilateral filtering***. All four techniques have a common basic principle, which is applying convolutional operations to the image with a filter (kernel). The values of the applying filters are different between the four blurring methods.

Average blurring is taking the average of all the pixel values under the given kernel area and replace the value at the center. For example, suppose we have a kernel with the size of 5X5. We calculate the average of the convoluted outcome and put that result to the center of the given area.

$$\frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \rightarrow$$

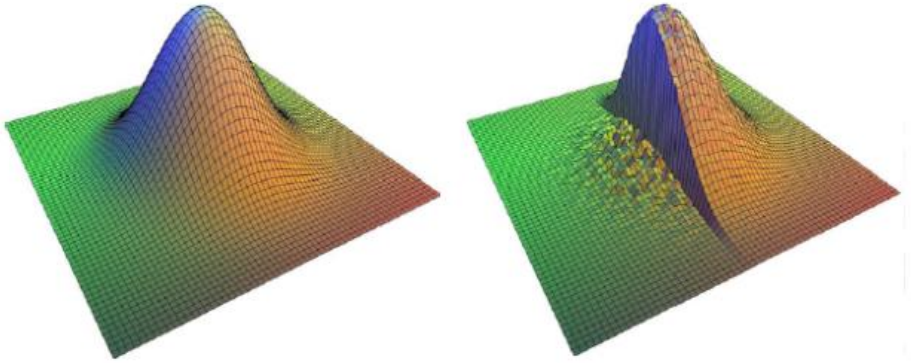
156	124	112	150	130	142
128	180	130	144	145	140
111	150	110	200	142	128
101	130	170	250	133	120
121	150	200	170	119	150
130	150	150	135	200	150

Then what will it be like if we increase the size of the kernel? As the size of filters gets bigger, the pixel values will be normalized more. Therefore we can expect the image to get blurred the more. Let's check out the result with the code as follows. (For comparison, I'll keep attaching the original image to the result)

```
# Import the image and convert to RGB
img = cv2.imread('text.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)# Plot the image with different kernel sizes
kernels = [5, 11, 17]fig, axs = plt.subplots(nrows = 1, ncols = 3, figsize = (20, 20))
for ind, s in enumerate(kernels):
    img_blurred = cv2.blur(img, ksize = (s, s))
    ax = axs[ind]
    ax.imshow(img_blurred)
    ax.axis('off')
plt.show()
```

Original Image (5, 5) kernel (11, 11) kernel (17, 17) kernel

Medium blurring is the same with average blurring except that it uses the median value instead of the average. Therefore when we have to handle sudden noises in the image such as '[salt and pepper noise](),' it'll be better to use medium blurring than average blurring.



[The shape of a Gaussian filter (on the left) and a Bilateral filter (on the right)]()

Gaussian blurring is nothing but using the kernel whose values have a Gaussian distribution. The values are generated by a Gaussian function so it requires a sigma value for its parameter. As you can see the image above, the values of the kernel go higher near the center and go smaller near the corner. It's good to apply this method to the [noises that have a normal distribution]() such as [white noise]().

Bilateral Filtering is an advanced version of Gaussian blurring. Blurring produces not only dissolving noises but also smoothing edges. And bilateral filter can keep edges sharp while removing noises. It uses Gaussian-distributed values but takes both distance and the pixel value differences into account. Therefore it requires `sigmaSpace` and `sigmaColor` for the parameters.

```
# Blur the image
img_0 = cv2.blur(img, ksize = (7, 7))
img_1 = cv2.GaussianBlur(img, ksize = (7, 7), sigmaX = 0)
```

```
img_2 = cv2.medianBlur(img, 7)
img_3 = cv2.bilateralFilter(img, 7, sigmaSpace = 75, sigmaColor =75)# Plot the images
images = [img_0, img_1, img_2, img_3]
fig, axs = plt.subplots(nrows = 1, ncols = 4, figsize = (20, 20))for ind, p in
enumerate(images):
    ax = axs[ind]
    ax.imshow(p)
    ax.axis('off')
plt.show()
```

Average Blur

Gaussian Blur

Median Blur

Bilateral Filtering

Thresholding

Thresholding transforms images into binary images. We need to set the threshold value and max values and then we convert the pixel values accordingly. There are five different types of thresholding: ***Binary, the inverse of Binary, Threshold to zero, the inverse of Threshold to Zero,*** *and* ***Threshold truncation***.

```
img = cv2.imread('gradation.png')# Thresholding
_, thresh_0 = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY)
_, thresh_1 = cv2.threshold(img, 127, 255, cv2.THRESH_BINARY_INV)
_, thresh_2 = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO)
_, thresh_3 = cv2.threshold(img, 127, 255, cv2.THRESH_TOZERO_INV)
_, thresh_4 = cv2.threshold(img, 127, 255, cv2.THRESH_TRUNC)# Plot the images
images = [img, thresh_0, thresh_1, thresh_2, thresh_3, thresh_4]fig, axs =
plt.subplots(nrows = 2, ncols = 3, figsize = (13, 13))
for ind, p in enumerate(images):
    ax = axs[ind//3, ind%3]
    ax.imshow(p)
plt.show()
```

$$\text{THRESH\_BINARY} = \begin{cases} \text{maxval} & I(x,y) > \text{thresh} \\ 0 & \text{else} \end{cases}$$

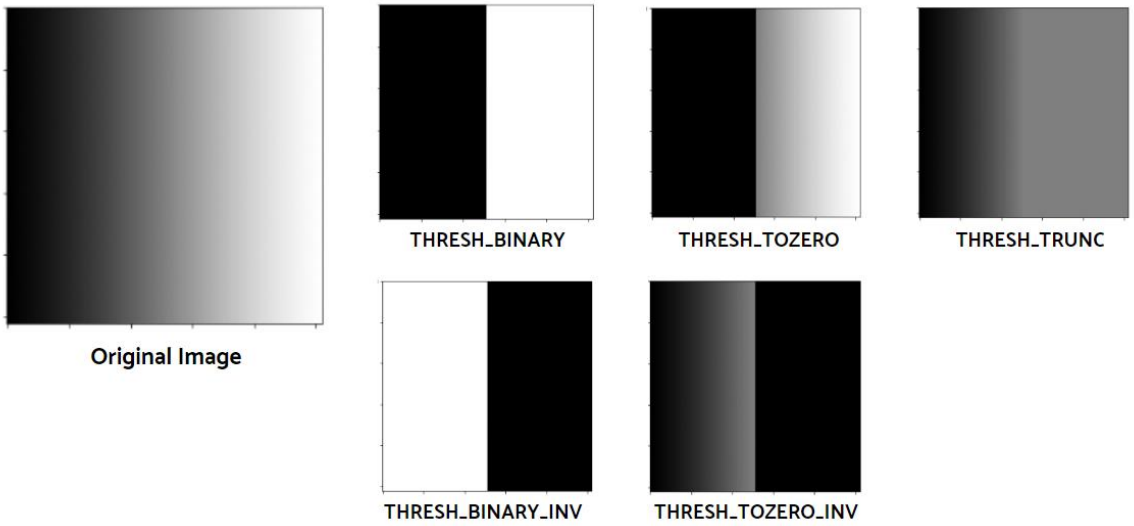
$$\text{THRESH\_BINARY\_INV} = \begin{cases} 0 & I(x,y) > \text{thresh} \\ \text{maxval} & \text{else} \end{cases}$$

$$\text{THRESH\_TOZERO} = \begin{cases} I(x,y) & I(x,y) > \text{thresh} \\ 0 & \text{else} \end{cases}$$

$$\text{THRESH\_TOZERO\_INV} = \begin{cases} 0 & I(x,y) > \text{thresh} \\ I(x,y) & \text{else} \end{cases}$$

$$\text{THRESH\_TRUNC} = \begin{cases} \text{thresh} & I(x,y) > \text{thresh} \\ I(x,y) & \text{else} \end{cases}$$

You can see how each type of thresholding can be expressed in a mathematical way and **I(x, y)** is the intensity at the point, or the pixel value at (x, y). But I prefer to understand the concept visually. Take a look at the r you to comprehend the difference between the types.picture on the right. The images are way better fo



But don't you think it's too harsh to take just one value of threshold and apply it to all parts of an image? What if we have a picture with various amount of lighting in different areas? In this case, applying one value to the whole image would be a bad choice. A better approach would be using different thresholds for each part of the image. There is another technique called ***Adaptive thresholding***, which serves this issue. By calculating the threshold within the neighborhood area of the image, we can achieve a better result from images with varying illumination.

```
# Convert the image to grayscale
img = cv2.imread('text.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)# Adaptive Thresholding
_, thresh_binary = cv2.threshold(img, thresh = 127, maxval = 255, type =
cv2.THRESH_BINARY)
adap_mean_2 = cv2.adaptiveThreshold(img, 255,
                                    cv2.ADAPTIVE_THRESH_MEAN_C,
                                    cv2.THRESH_BINARY, 7, 2)
adap_mean_2_inv = cv2.adaptiveThreshold(img, 255,
                                        cv2.ADAPTIVE_THRESH_MEAN_C,
                                        cv2.THRESH_BINARY_INV, 7, 2)
adap_mean_8 = cv2.adaptiveThreshold(img, 255,
                                    cv2.ADAPTIVE_THRESH_MEAN_C,
                                    cv2.THRESH_BINARY, 7, 8)
adap_gaussian_8 = cv2.adaptiveThreshold(img, 255,
                                    cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                    cv2.THRESH_BINARY, 7, 8)
```

We need to convert the color mode to grayscale to apply adaptive thresholding. The parameters of adaptive thresholding are `maxValue` (which I set 255 above), `adaptiveMethod`, `thresholdType`, `blockSize` and `C`. And the adaptive method here has two kinds: `ADAPTIVE_THRESH_MEAN_C`, `ADAPTIVE_THRESH_GAUSSIAN_C`. Let's just see how images are produced differently.

```
# Plot the images
images = [img, thresh_binary, adap_mean_2, adap_mean_2_inv,
          adap_mean_8, adap_gaussian_8]
fig, axs = plt.subplots(nrows = 2, ncols = 3, figsize = (15, 15))for ind, p in
enumerate(images):
    ax = axs[ind%2, ind//2]
    ax.imshow(p, cmap = 'gray')
```

```
    ax.axis('off')
plt.show()
```


Original Image


Adaptive Threshold Mean
C = 2


Adaptive Threshold Mean
C = 8


THRESH_BINARY


Adaptive Threshold Mean
Inverse. C = 2


Adaptive Threshold Gaussian
C = 8

We have the original image and the one with binary thresholding on the left line. Compare this with the second and third image on the upper line, which is produced by `ADAPTIVE_THRESH_MEAN_C` . It shows a more detailed result than that of binary thresholding. We can also see that it gets more explicit when the C value is bigger. `C` indicates how much we'll subtract from the mean or weighted mean. With the two images on the right line, we can also compare the effect of `ADAPTIVE_THRESH_MEAN_C` and `ADAPTIVE_THRESH_GAUSSIAN_C` with the same C value.

Gradient

I believe we are already familiar with the concept of gradients. In mathematics, the gradient geometrically represents the slope of the graph of a function with multi-variables. As it is a vector-valued function, it takes a direction and a magnitude as its components. Here we can also bring the same concept to the pixel values of images as well. The image gradient represents directional changes in the intensity or color mode and we can use this concept for locating edges.

```
# Apply gradient filtering
sobel_x = cv2.Sobel(img, cv2.CV_64F, dx = 1, dy = 0, ksize = 5)
sobel_y = cv2.Sobel(img, cv2.CV_64F, dx = 0, dy = 1, ksize = 5)
blended = cv2.addWeighted(src1=sobel_x, alpha=0.5, src2=sobel_y,
                          beta=0.5, gamma=0)
laplacian = cv2.Laplacian(img, cv2.CV_64F)
```

Sobel operation uses both Gaussian smoothing and differentiation. We apply it by `cv2.Sobel()` and two different directions are available: vertical (`sobel_x`) and horizontal (`sobel_y`). `dx` and `dy` indicates the derivatives. When `dx = 1` , the operator calculates the derivatives of the pixel values along the horizontal direction to make a filter.

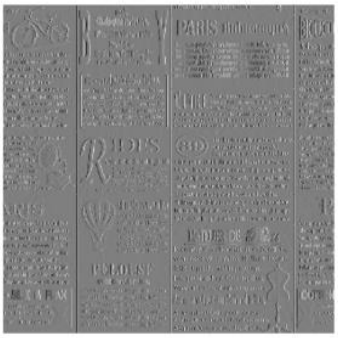
We can also apply in both directions by summing the two filters of `sobel_x` and `sobel_y` . With the function `cv2.addWeighted()` , we can calculate the weighted sum of the filters. As you can see above in the code cell, I gave the same amount of weight to the two filters.

Laplacian operation uses the second derivatives of x and y. The mathematical expression is shown below.

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

A picture is worth a thousand words. Let's see how the images are like.

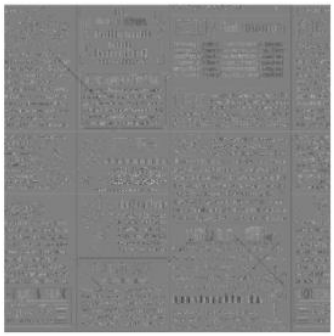
```
# Plot the images
images = [sobel_x, sobel_y, blended, laplacian]
plt.figure(figsize = (20, 20))
for i in range(4):
    plt.subplot(1, 4, i+1)
    plt.imshow(images[i], cmap = 'gray')
    plt.axis('off')
plt.show()
```



Sobel X

Sobel Y

Sobel X + Sobel Y



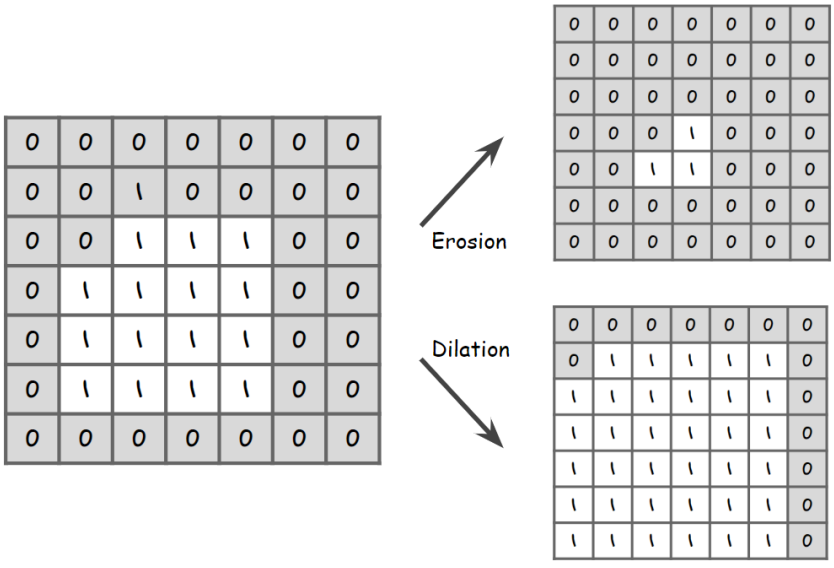
Laplacian

It's apparent that the first and second image have a directional pattern. With the first image, we can clearly see the edges in the vertical direction. With the second image, we can see the horizontal edges. And both the third and fourth images, the edges on both directions are shown.

Morphological transformations

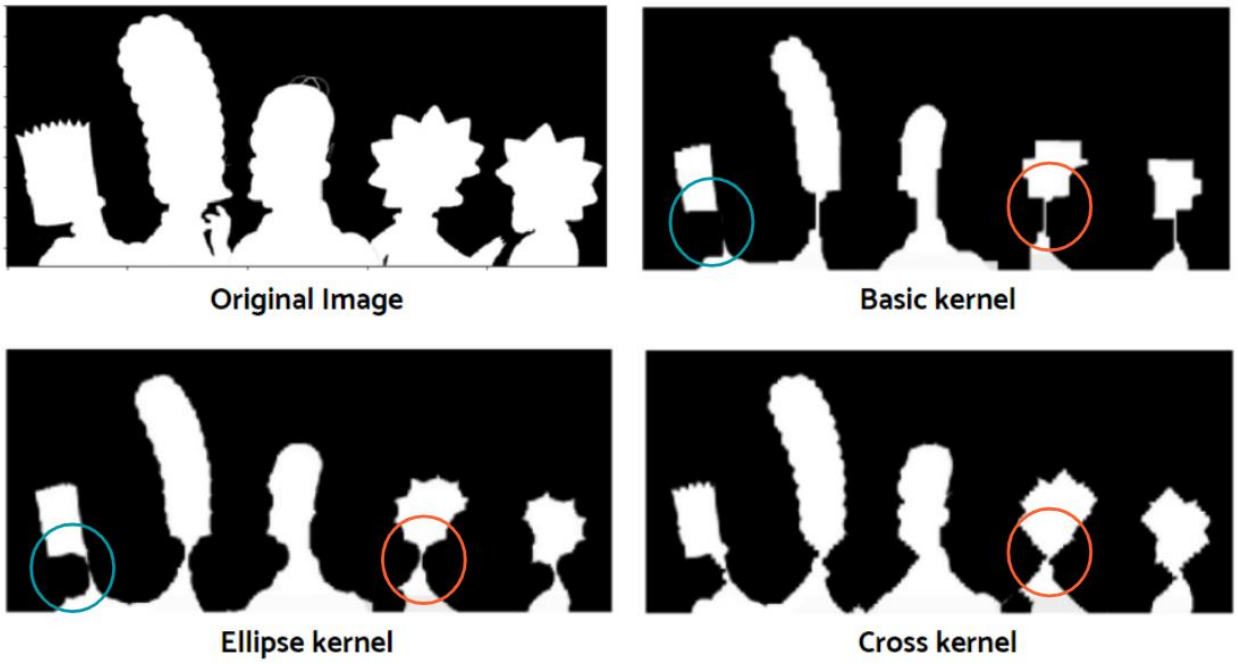
It's also possible to manipulate the figures of images by filtering, which is called as ***morphological transformation***. Let's talk about erosion and dilation first.

Erosion is the technique for shrinking figures and it's usually processed in a grayscale. The shape of filters can be a rectangle, an ellipse, and a cross shape. By applying a filter we remove any 0 values under the given area.



Let's see how these can be implemented in codes.

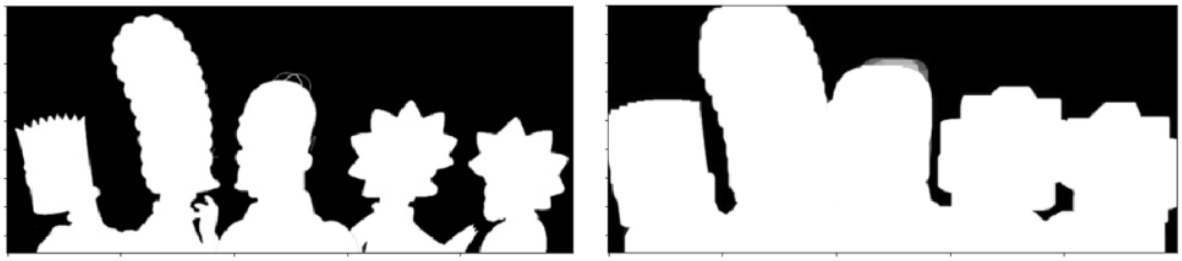
```
img = cv2.imread('simpson.jpg')# Create erosion kernels
kernel_0 = np.ones((9, 9), np.uint8)
kernel_1 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (9, 9))
kernel_2 = cv2.getStructuringElement(cv2.MORPH_CROSS, (9, 9))kernels = [kernel_0,
kernel_1, kernel_2]# Plot the images
plt.figure(figsize = (20, 20))
for i in range(3):
    img_copy = img.copy()
    img_copy = cv2.erode(img_copy, kernels[i], iterations = 3)
    plt.subplot(1, 3, i+1)
    plt.imshow(img_copy)
    plt.axis('off')
plt.show()
```



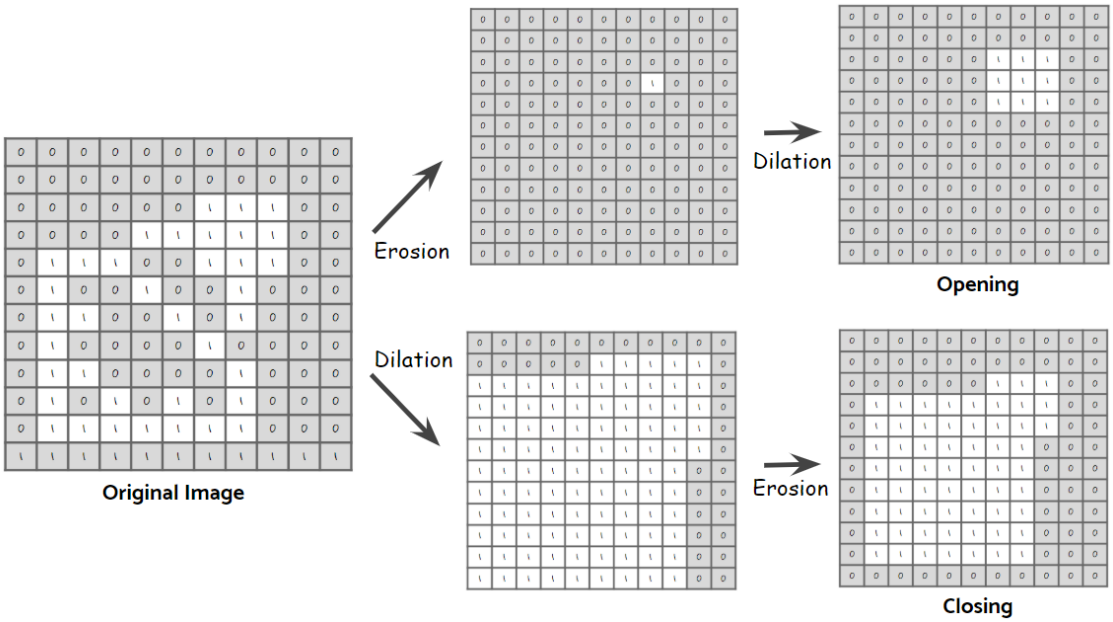
Check out how the Simpson's family shrank with different types of kernels applied. (Sorry to Simpson for losing his hands!) We can see the image with the ellipse filter is eroded in a "round" way while the one with the basic filter with a squared shape is eroded in a "linear" way. The last one with the cross filter shows it shrank in a "diagonal" way.

Dilation is the opposite to erosion. It is making objects expand and the operation will be also opposite to that of erosion. Let's check out the result with the code as follows.

```
# Apply dilation
kernel = np.ones((9, 9), np.uint8)
img_dilate = cv2.dilate(img, kernel, iterations = 3)plt.figure(figsize = (20, 10))
plt.subplot(1, 2, 1); plt.imshow(img, cmap="gray")
plt.subplot(1, 2, 2); plt.imshow(img_dilate, cmap="gray")
plt.show()
```



Opening and **closing** operation is the mixed version of erosion and dilation. Opening performs erosion first and then dilation is performed on the result from the erosion while closing performs dilation first and the erosion.



As you can see the picture above, closing is useful to detect the overall contour of a figure and opening is suitable to detect subpatterns. We can implement these operators with the function `cv2.morphologyEx()` shown below. The parameter `op` indicates which type of operator we're going to use.

```
# Apply the operations
kernel = np.ones((9, 9), np.uint8)img_open = cv2.morphologyEx(img, op= cv2.MORPH_OPEN,
kernel)
img_close = cv2.morphologyEx(img, op= cv2.MORPH_CLOSE, kernel)
img_grad = cv2.morphologyEx(img, op= cv2.MORPH_GRADIENT, kernel)
img_tophat = cv2.morphologyEx(img, op= cv2.MORPH_TOPHAT, kernel)
img_blackhat = cv2.morphologyEx(img, op= cv2.MORPH_BLACKHAT, kernel)# Plot the images
images = [img, img_open, img_close, img_grad,
          img_tophat, img_blackhat]fig, axs = plt.subplots(nrows = 2, ncols = 3, figsize
= (15, 15))
for ind, p in enumerate(images):
    ax = axs[ind//3, ind%3]
    ax.imshow(p, cmap = 'gray')
    ax.axis('off')
plt.show()
```

Original Image

Opening

Closing

Gradient

Tophat

Blackhot

Note that the Simpson's hand is depicted differently in the images with the opening filter and closing filter. Gradient filter (`MORPH_CGRADIENT`) is the subtracted area from dilation to erosion. Top hat filter (`MORPH_TOPHAT`) is the subtracted area from opening to the original image while black hot filter (`MORPH_BLACKHAT`) is that from closing. I'd recommend you to visit **here** to get further explanations on morphological operators.

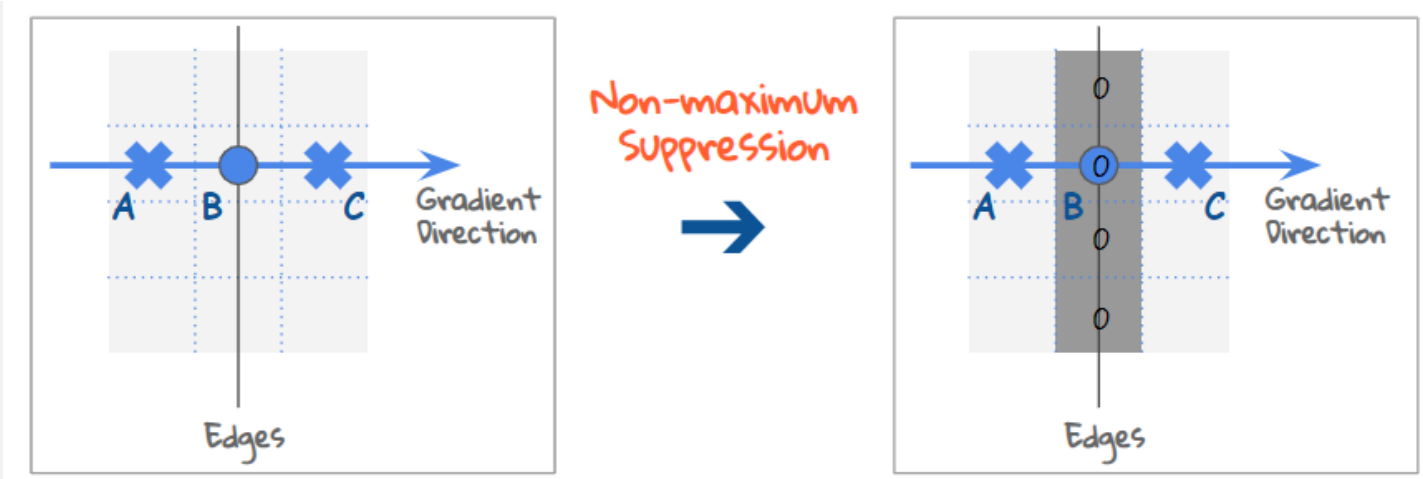
From Feature Detection to Face DetectionD

Detection task is one of the major tasks for computer vision, and there are so many ways we can utilize this technique. Identifying an error that can be critical when it occurs but can't be recognizable by human's eyes. Detecting an unsafe and risky motion or moment to save a life. Perceiving spatial information for a self-driving car. The examples of Object detection usage are countless and making those tasks automatic will bring us safety as well as efficiency.

Edge Detection

Edge detection means identifying points in an image where the brightness changes sharply or discontinuously. We can draw line segments with those points, which are called ***edges***. Actually, we already learned one of the edge detection techniques last time. Do you remember? Gradient filtering with Sobel and Laplacian operation. By calculating the derivatives of pixel values in a given direction, gradient filtering can depict the edges of images.

Canny detection is another type of edge detection techniques. It's one of the most popular algorithms for detecting edges, which is performed in four steps: ***Noise reduction, Finding gradient and its direction, Non-maximum suppression*** and ***hysteresis thresholding***. The algorithm starts with Gaussian blurring and I guess we already know the reason for removing noises in images. And then it finds the intensity gradient of the image with a Sobel kernel. With the derived gradient and direction, every pixel is checked if a certain point is a local maximum in its surrounding points. If it's not, this point is suppressed to zero (total absence, black). This is called ***non-maximum suppression***.

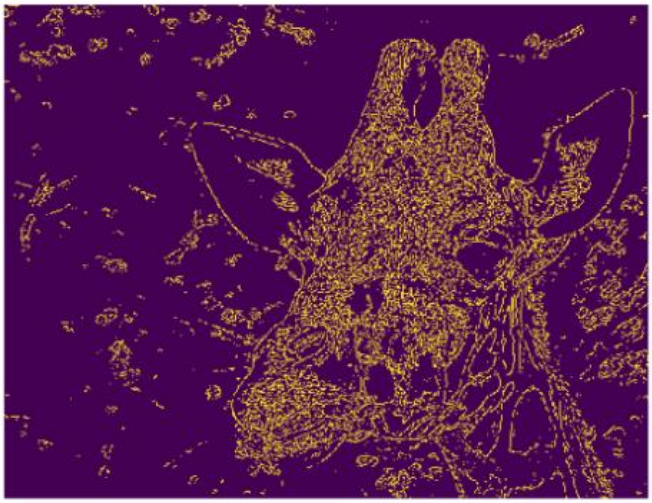


If the point is considered to be a local maximum, it goes to the next stage. The final stage is the last decision stage whether the edges at the previous steps are really edges or not. This is called ***Hysteresis Thresholding*** and we need two threshold values here.

Given the two different threshold values, we get three ranges of values. So if the intensity gradient of a point is higher than the upper threshold, it will be considered as 'sure-edge.' If the gradient of a point is lower than the lower threshold, the point will be discarded. And in case of the gradient being in the middle of the two thresholds, we see its connectivity to other 'sure-edge' points. If there's no connection, it will be discarded as well.

```
img = cv2.imread('images/giraffe.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)# Canny detection without blurring
edges = cv2.Canny(image=img, threshold1=127,
threshold2=127)plt.figure(figsize = (20, 20))
plt.subplot(1, 2, 1); plt.imshow(img)
plt.axis('off')
plt.subplot(1, 2, 2); plt.imshow(edges)
plt.axis('off')
```

Original Image



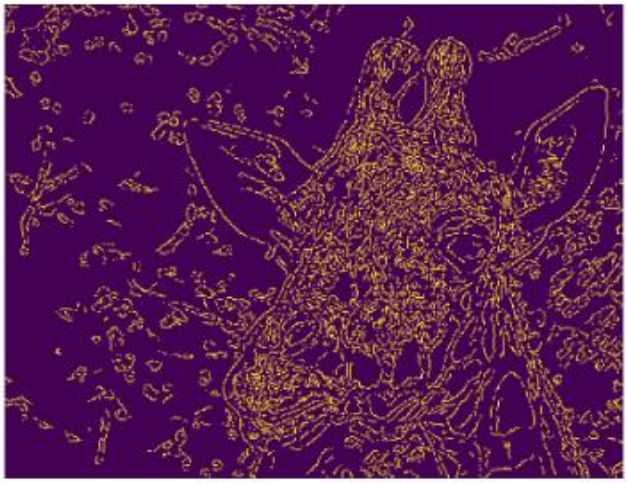
Canny without Blur

I just used the median value for the two thresholds without blurring and the result isn't quite desirable. Now let's try different threshold values this time.

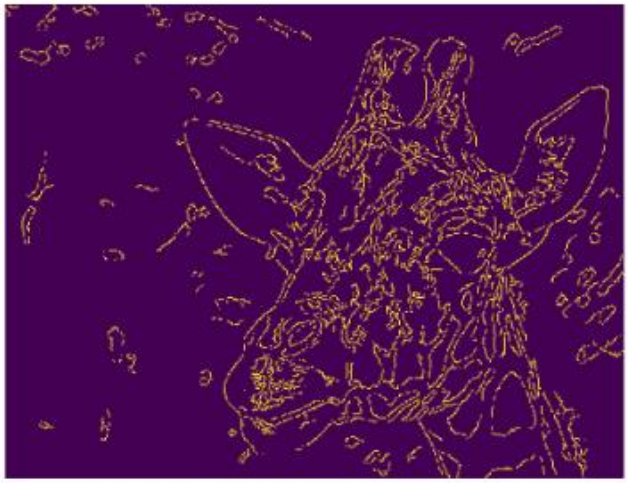
```
# Set the lower and upper threshold
med_val = np.median(img)lower = int(max(0, .7*med_val))
upper = int(min(255, 1.3*med_val))
```

To see how blurring can change the outcome, I'm going to apply two different sizes of kernels, (5x5) and (9x9). And I'm going to try changing the upper threshold values by adding 100. So we have 4 types of processed images as follows:

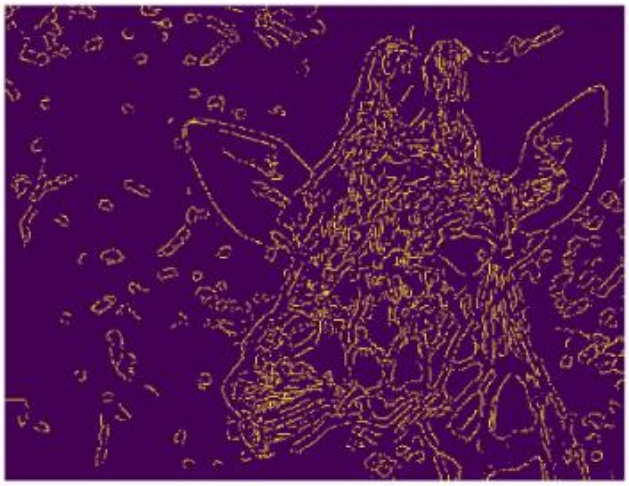
```
# Blurring with ksize = 5
img_k5 = cv2.blur(img, ksize = (5, 5))# Canny detection with different
thresholds
edges_k5 = cv2.Canny(img_k5, threshold1 = lower, threshold2 = upper)
edges_k5_2 = cv2.Canny(img_k5, lower, upper+100)# Blurring with ksize = 9
img_k9 = cv2.blur(img, ksize = (9, 9))# Canny detection with different
thresholds
edges_k9 = cv2.Canny(img_k9, lower, upper)
edges_k9_2 = cv2.Canny(img_k9, lower, upper+100)# Plot the images
images = [edges_k5, edges_k5_2, edges_k9, edges_k9_2]
plt.figure(figsize = (20, 15))
for i in range(4):
    plt.subplot(2, 2, i+1)
    plt.imshow(images[i])
    plt.axis('off')
plt.show()
```



ksize = (5, 5)
lower, upper



ksize = (5, 5)
lower, upper + 100



ksize = (9, 9)
lower, upper

ksize = (9, 9)
lower, upper + 100

As you can see above, blurring helps to remove noises and we got the better result with (9x9) size of the kernel. Also, we got a better result with the higher upper threshold value.

Corner Detection

Corner detection is another detection algorithm which is widely used in object detection, motion detection, video tracking and so on. What is a corner in image processing? How can we define a corner with pixels? We see a corner as a junction where edges intersect. Then how can we find them? Finding all edges first and then locating the points where they cross each other? Actually we have another way of making things more efficient, which are ***Harris corner detection*** and ***Shi & Tomasi corner detection***.

These algorithms work as follows. We detect points where there's a considerable change in their intensity values in all directions. And then we construct a matrix to extract eigenvalues from it. These eigenvalues are for the sake of scoring points to decide if it's a corner or not. The mathematical expressions are shown below.

$$I(x+\Delta x, y+\Delta y) \approx I(x,y) + I_x(x,y)\Delta x + I_y(x,y)\Delta y$$

$$f(x,y) = \sum_{(x_k,y_k)\in W} (I(x_k,y_k) - I(x_k+\Delta x, y_k+\Delta y))^2$$

$$\approx \sum_{(x,y)\in W} (I_x(x,y)\Delta x + I_y(x,y)\Delta y)^2$$

$$\approx (\Delta x, \Delta y)\, M \begin{pmatrix}\Delta x \\ \Delta y\end{pmatrix}$$

Structure tensor

$$M = \sum_{(x,y)\in W} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$ → eigenvalues of M λ_1, λ_2

① **Harris Corner Detection :** $R = \lambda_1\lambda_2 - k(\lambda_1+\lambda_2)^2 = det(M) - k(trace(M))^2$

② **Shi & Tomasi Corner Detection:** $R = \min(\lambda_1, \lambda_2)$

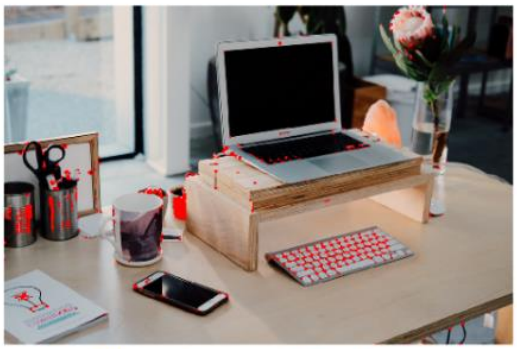
Now let's see how we can implement these with codes. We first need to convert the image into grayscale. **Harris corner detection** can be performed with a function `cv2.cornerHarris()` in OpenCV.

```
img = cv2.imread('images/desk.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)# Apply Harris corner
detection
dst = cv2.cornerHarris(img_gray, blockSize = 2, ksize = 3, k = .04)
```

The parameter `blockSize` is the size of the window to consider as neighborhood and `k` is Harris detector free parameter which is shown in the equation above. The result is the score R and we'll use this to detect corners.

```
# Spot the detected corners
img_2 = img.copy()
img_2[dst>0.01*dst.max()]=[255,0,0]# Plot the image
plt.figure(figsize = (20, 20))
plt.subplot(1, 2, 1); plt.imshow(img)
plt.axis('off')
plt.subplot(1, 2, 2); plt.imshow(img_2)
plt.axis('off')
```

Original Image



Harris Corner Detection

Let's try **Shi-Tomasi corner detection** this time. We can use this with a function `cv2.goodFeaturesToTrack()`. We set the maximum number of corners sorted by the most likelihood (`maxCorners`). We also assign minimum distance (`minDistance`) and minimum quality level (`qualityLevel`) that is required to be considered as corners. After we get the detected corners, we'll mark those points with circles as follows.

```
# Apply Shi-Tomasi corner detection
corners = cv2.goodFeaturesToTrack(img_gray, maxCorners = 50,
                                  qualityLevel = 0.01,
                                  minDistance = 10)
corners = np.int0(corners)# Spot the detected corners
img_2 = img.copy()
for i in corners:
    x,y = i.ravel()
    cv2.circle(img_2, center = (x, y),
               radius = 5, color = 255, thickness = -1)# Plot the image
plt.figure(figsize = (20, 20))
plt.subplot(1, 2, 1); plt.imshow(img)
plt.axis('off')
plt.subplot(1, 2, 2); plt.imshow(img_2)
plt.axis('off')
```

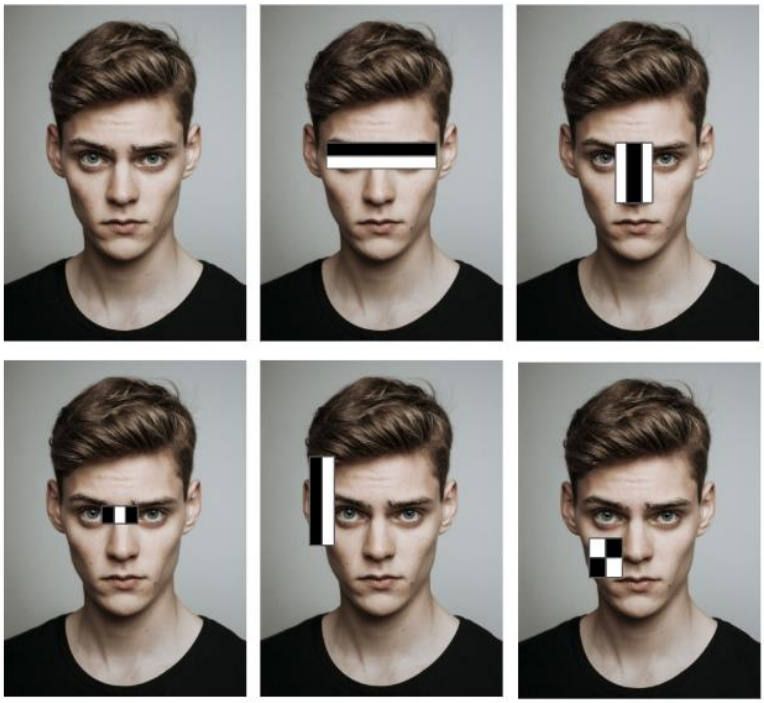
Original Image

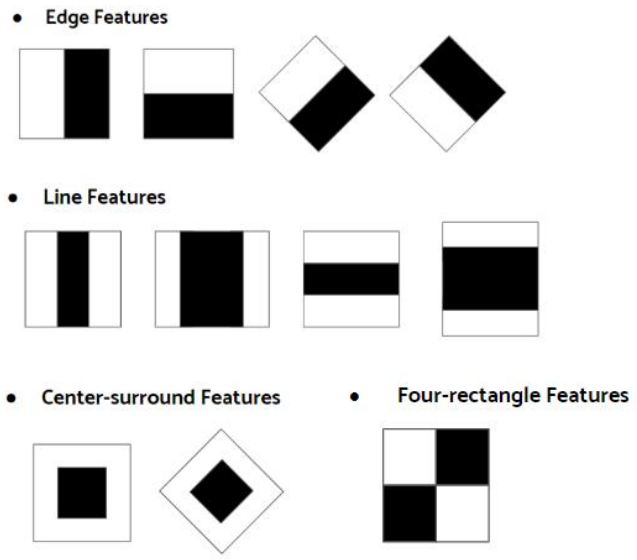
Shi-Tomasi Corner Detection

Face Detection

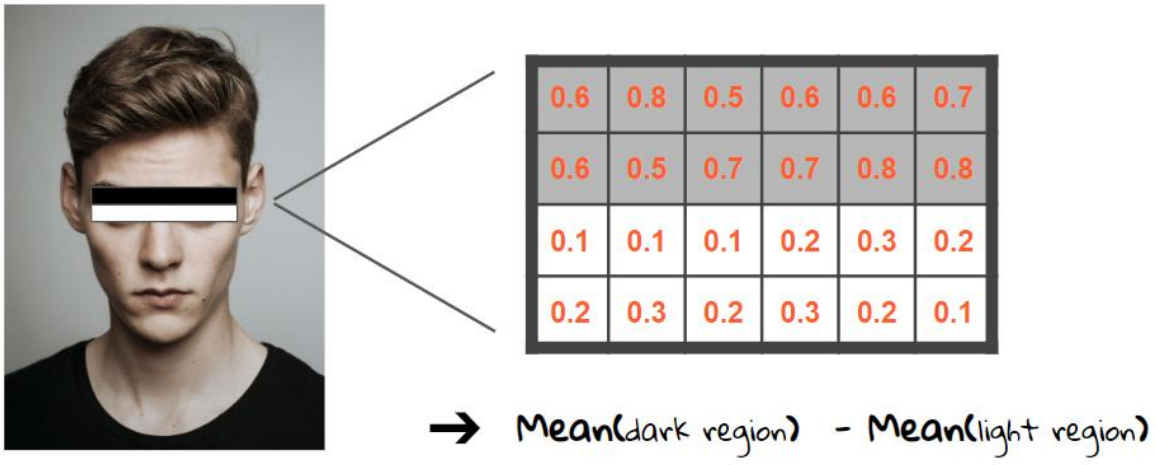
The last feature we're going to see is a face. **Face detection** is a technology identifying the presence and the position of human faces in digital images. I want you to differentiate face detection from **face recognition** which indicates detecting the identification of a person by his or her face. So face detection can't tell us to whom the detected face belongs.

Face detection is basically a classification task so it's trained to classify whether there is a target object or not. And ***Haar Feature-based Cascade Classifier*** is one of the face detection models available in OpenCV. This is a pre-trained model, which means it already completed training with thousands of images. The 4 key points for understanding this algorithm are ***Haar features extraction, integral image, Adaboost*** and ***cascade classifiers***.



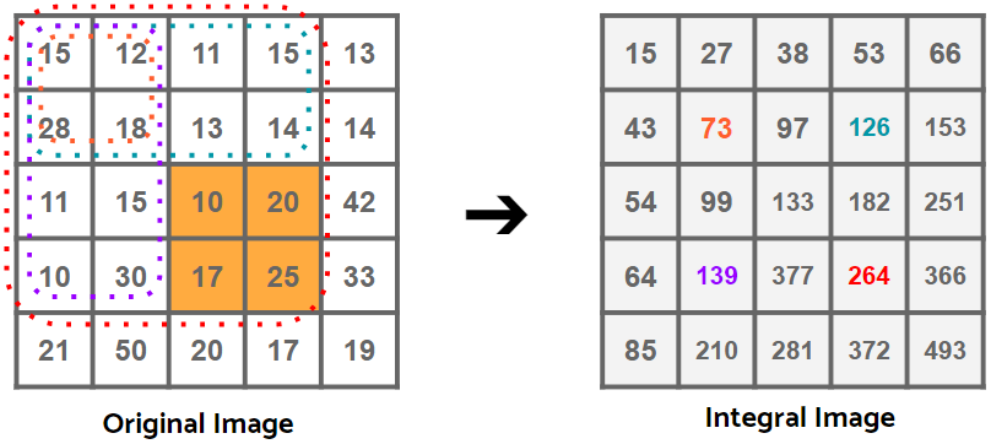


Haar-like features are image filters or image kernels used in object detection and the examples are shown above. They owe their name to their intuitive similarity with Haar wavelets which is originally proposed by Alfréd Haar. During detection, we pass the window on an image and do the convolutional operation with the filters to see if there's the feature we're looking for is in the image. Here is **the video** which visualizes how the detection works.

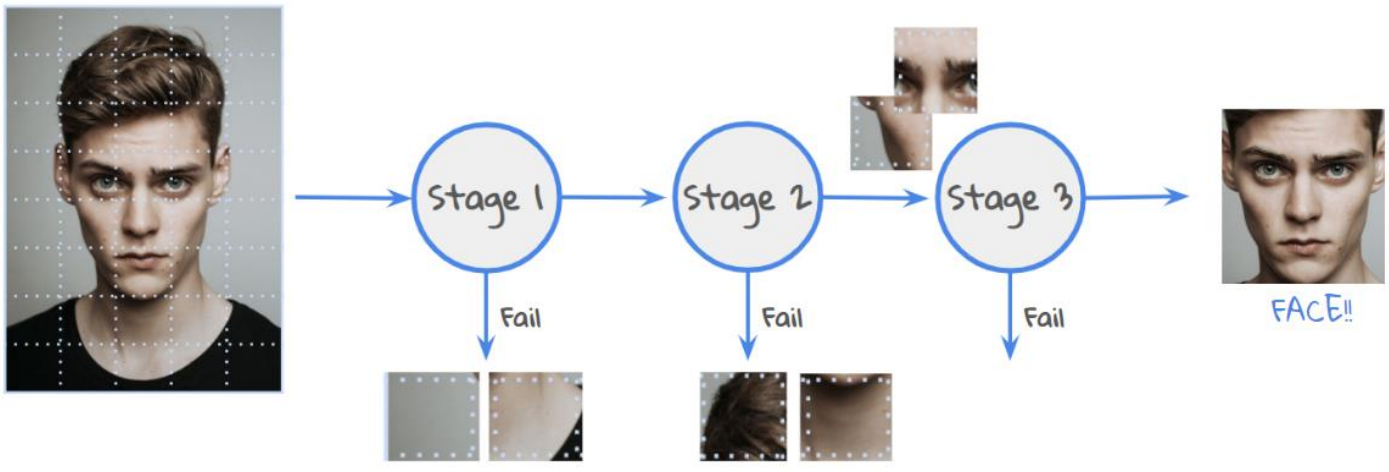


So how we decide whether there is a wanted feature or not in a given area? Let's take a look at the picture above. We have a kernel whose upper half is dark and lower half is light. Then we get the mean of the pixel values for each region and subtract the gap between the two. If the result is higher than a threshold, say 0.5, then we conclude there's the feature we're detecting. We repeat this process for each kernel while sliding the window over the image.

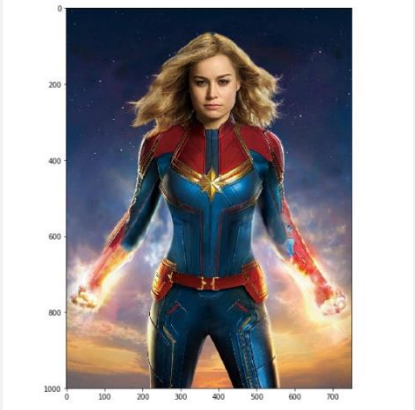
Although this isn't a complex calculation, the total amount of computation becomes huge when we consider it in a whole image. If you've seen the video mentioned above, you'd get the intuition into the amount of calculation involved. And this is where an ***integral image*** comes into play. The integral image is a way of image representation which is derived to make the feature evaluation faster and more effective.
As you can see below, there are the pixels of an image on the left and an integral image on the right. Starting from the left upper point, it computes the accumulated sum of pixels under a given rectangular area. On the integral image, the sum of the pixels in the box with the dashed line is written at the lower right point of the box on the right.



With this pre-calculated table, we can simply get the summed value for a certain area by the values of sub-rectangles (the red, orange, blue and purple box).
So we solved the computational cost with the integral image. But we're not done yet. Think about when the detecting window is at the blank background where there's no object or face. It'll still be a waste of time if it performs the same process in such a part. And there's another kick making this detector faster. Implementing ***a cascade classifier*** with Adaboost!

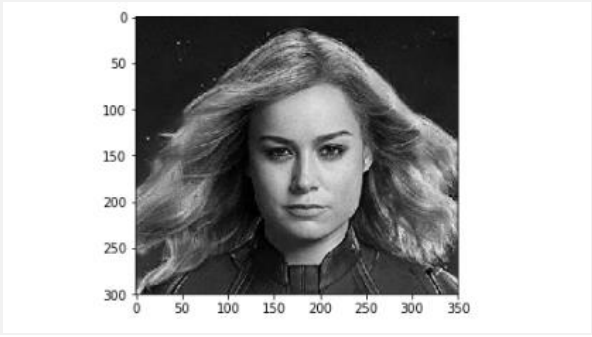


A cascade classifier constructs stepwise stages and gives an order among Haar-like features. Basic forms of the features are implemented at the early stages and the more complex ones are applied only for those promising regions. And at each stage, the Adaboost model will be trained by ensembling weak learners. If a subpart, or a sub-window, is classified as 'not a face-like region' at the prior stage, it's rejected to the next step. By doing so, we can only consider the survived ones and achieve much higher speed.



We're going to use only the part of this image. So let's get the region of interest around her face first and then convert the image into grayscale. The reason for using only one channel is because we're only interested in the changes in the light intensity of the features.

```
cap_mavl = cv2.imread('images/captin_marvel.jpg')# Find the region of
interest
roi = cap_mavl[50:350, 200:550]
roi = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
plt.imshow(roi, cmap = 'gray')
```



Haar Cascade classifier files basically come along with OpenCV. You can probably find them in the OpenCV folder on your computer. Or you can simply download the file **here**.

```
# Load Cascade filter
face_cascade =
cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_default.xml')
```

Next, we're going to create a function detecting a face and drawing a rectangle around it. To detect the face, we can use the method `.detectMulitiScale()` of the classifier `face_cascade` that we loaded above. It returns the four points of the identified region so we'll draw a rectangle at that position. `scaleFactor` is a parameter for how much the image size is reduced at each image scale and `minNeighbors` for how many neighbors each candidate rectangle should be trained. Now let's apply this function to the image and see the result.

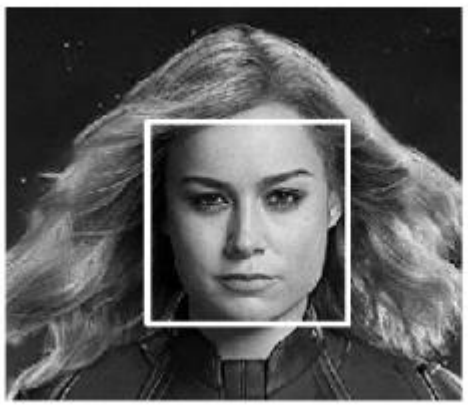
```
# Create the face detecting function
def detect_face(img):

    img_2 = img.copy()
```

```
    face_rects = face_cascade.detectMultiScale(img_copy,
                                               scaleFactor = 1.1,
                                               minNeighbors = 3)

    for (x, y, w, h) in face_rects:
        cv2.rectangle(img_2, (x, y), (x+w, y+h), (255, 255, 255), 3)

    return img_2# Detect the face
roi_detected = detect_face(roi)
plt.imshow(roi_detected, cmap = 'gray')
plt.axis('off')
```



Great! I think this is quite satisfactory. Why don't we call other heroes altogether this time? We can implement this classifier to an image with multiple faces as well.

```
# Load the image file and convert the color mode
avengers = cv2.imread('images/avengers.jpg')
avengers = cv2.cvtColor(avengers, cv2.COLOR_BGR2GRAY)# Detect the face and
plot the result
detected_avengers = detect_face(avengers)
display(detected_avengers, cmap = 'gray')
```

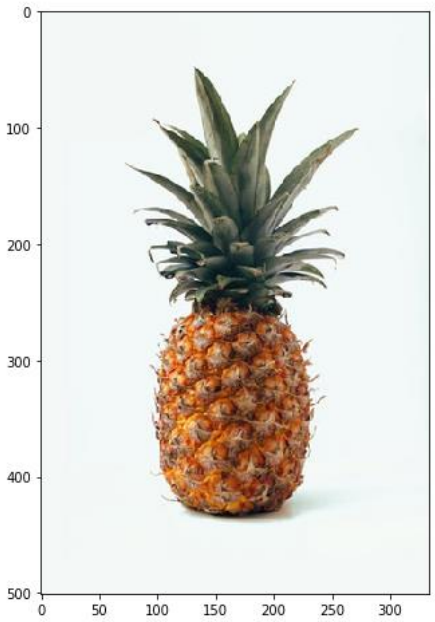
And yes, it sometimes makes a failure catching a 'non-face' object or missing a 'real-face.' It's funny that it detected Spider-Man successfully while mistook the hands of Captin America and Black Widow as eyes. We usually get better results with a face looking in front and showing its forehead and eyes clearly.

Contour detection

You may be already familiar with the word 'contour.' I've used this term several times in previous posts. **A contour line** indicates a curved line representing the boundary of the same values or the same intensities. A contour map is the most straightforward example we can think of.

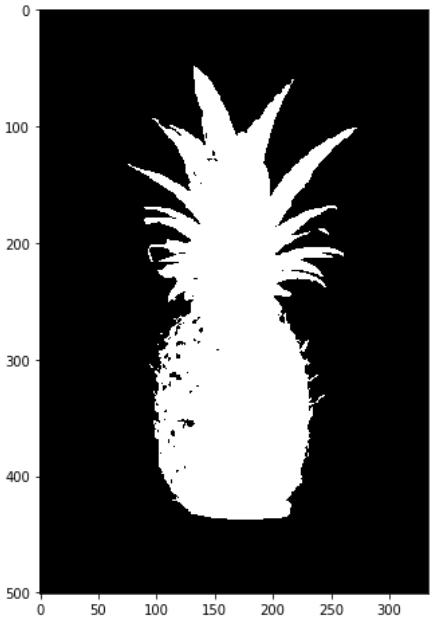
But then you may ask this. What's the difference between edges and contours? The two terms are often used interchangeably so it could be a bit confusing. To put it simply, the concept of edges lies in a local range while the concept of contours is at the overall boundary of a figure. Edges are points whose values change significantly compared to their neighboring points. Contours, on the other hand, are closed curves which are obtained from edges and depicting a boundary of figures. You can find a further explanation **here**.

```
# Load the image
img = cv2.imread('images/pine_apple.jpg')
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(img)
```



So what we're going to do is detecting the contour of this pineapple. Before applying the detection algorithm, we need to convert the image into grayscale and apply thresholding as follows. All of these steps are what we've discussed in the previous series.

```
# Blurring for removing the noise
img_blur = cv2.bilateralFilter(img, d = 7,
                               sigmaSpace = 75, sigmaColor =75)# Convert to grayscale
img_gray = cv2.cvtColor(img_blur, cv2.COLOR_RGB2GRAY)# Apply the thresholding
a = img_gray.max()
_, thresh = cv2.threshold(img_gray, a/2+60, a,cv2.THRESH_BINARY_INV)
plt.imshow(thresh, cmap = 'gray')
```



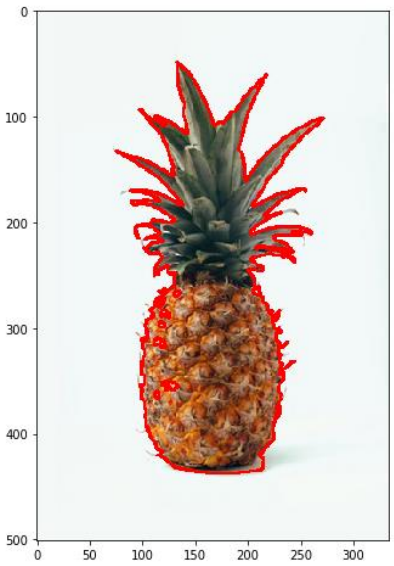
Contour detection can be implemented by the function `cv2.findContours()` in OpenCV and there are two important parameters here. `mode` is the way of finding contours, and `method` is the approximation method for the detection. I ask you to find other information from **the documentation**.

```
# Find the contour of the figure
image, contours, hierarchy = cv2.findContours(
                                    image = thresh,
                                    mode = cv2.RETR_TREE,
                                    method = cv2.CHAIN_APPROX_SIMPLE)
```

The mode `cv2.RETR_TREE` finds all the promising contour lines and reconstructs a full hierarchy of nested contours. The method `cv2.CHAIN_APPROX_SIMPLE` returns only the endpoints that are necessary for drawing the contour line. And as you can see above, this function gives the image, the detected contours and their hierarchy as its output.

The returned contour is a list of points consisting of the contour lines. To draw the outer line of the figure, we'll sort the contours by their area. With the selected contour line, `cv2.drawContours()` will depict the bounding line along with the points as shown below.

```
# Sort the contours
contours = sorted(contours, key = cv2.contourArea, reverse = True)# Draw the contour
img_copy = img.copy()
final = cv2.drawContours(img_copy, contours, contourIdx = -1,
                         color = (255, 0, 0), thickness = 2)
plt.imshow(img_copy)
```



It's simple, right? Although the shadow is also added at the bottom, the outcome is pretty satisfactory.

More about contours

There are more things we can do with the contour. We can find the centroid of an image or calculate the area of a boundary field with the help of the notion called image moment. The word *'moment'* is a short period of time in common usage. But in physics terminology, a **moment** is the product of the distance and another physical quantity meaning how a physical quantity is distributed or located. So in computer vision, **Image moment** is how image pixel intensities are distributed according to their location. It's a weighted average of image pixel intensities and we can get the centroid or spatial information from the image moment.

- **Spatial Moments**

$$M_{ij} = \sum_x \sum_y x^i y^j I(x,y)$$ → m00, m01, m02, m03, m10, m11, m12, m20, m21, m30

- **Central Moments**

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q I(x,y)$$ → mu02, mu03, mu11, mu12, mu20, mu21, mu30

- Centroid point : $\bar{x} = \frac{M_{10}}{M_{00}}$, $\bar{y} = \frac{M_{01}}{M_{00}}$

- **Normalized Central Moments**

$$\eta_{ij} = \frac{\mu_{ij}}{\mu_{00}^{(1+\frac{i+j}{2})}}$$ → nu02, nu03, nu11, nu12, nu20, nu21, nu30

There are three types of moments- spatial moments, central moments, and central normalized moments. We can get the image moment with the function `cv2.moments()` in OpenCV and it returns 24 different moments. If you print the output `M` as shown below, it'll return the 24 moments in a dictionary format.

```
# The first order of the contours
c_0 = contours[0]# image moment
M = cv2.moments(c_0)
print(M.keys())
```

I'd like to focus on the implementation of the image moments here. Additional reading resources can be found at the end of this article if you may have an interest.

To get the area of the contours, we can implement the function `cv2.contourArea()` . Why don't we try several contours here? If you input the first, second and third contours, you'll get the decreasing values as shown below. This shows that the contour detection algorithm forms the hierarchy of the detected boundaries.

```
# The area of contours
print("1st Contour Area : ", cv2.contourArea(contours[0])) # 37544.5
print("2nd Contour Area : ", cv2.contourArea(contours[1])) # 75.0
print("3rd Contour Area : ", cv2.contourArea(contours[2])) # 54.0
```

The arc length of the contour can be obtained by the function `cv2.arcLength()` . The parameter `closed` indicates whether the curve should be closed or not.

```
# The arc length of contours
print(cv2.arcLength(contours[0], closed = True))      # 2473.3190
print(cv2.arcLength(contours[0], closed = False))     # 2472.3190
```

Now let's try plotting the centroid and the extreme points of our pineapple. We can get the centroid point with the formula as follows.

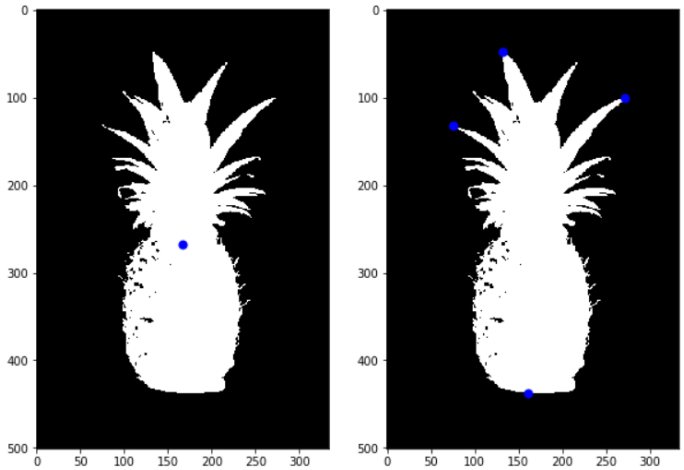
```
# The centroid point
cx = int(M['m10'] / M['m00'])
cy = int(M['m01'] / M['m00'])
```

The extrema are the endpoints on the left and right, at the top and bottom. And we can arrange x and y coordinates separately as follows.

```
# The extreme points
l_m = tuple(c_0[c_0[:, :, 0].argmin()][0])
r_m = tuple(c_0[c_0[:, :, 0].argmax()][0])
t_m = tuple(c_0[c_0[:, :, 1].argmin()][0])
b_m = tuple(c_0[c_0[:, :, 1].argmax()][0])pst = [l_m, r_m, t_m, b_m]
xcor = [p[0] for p in pst]
ycor = [p[1] for p in pst]
```

Now let's plot all these points on the image.

```
# Plot the points
plt.figure(figsize = (10, 16))
plt.subplot(1, 2, 1)
plt.imshow(image, cmap = 'gray')
plt.scatter([cx], [cy], c = 'b', s = 50)plt.subplot(1, 2, 2)
plt.imshow(image, cmap = 'gray')
plt.scatter(xcor, ycor, c = 'b', s = 50)
```



Other shapes of contours

Besides the compact contour, we can also draw a convex contour or rectangular contour line of a figure. Let's try a straight rectangular shape first. With the outer contour line, we'll draw the rectangle around the object. The function `cv2.boundingRect()` returns the 4 points of the bounding box as shown below.

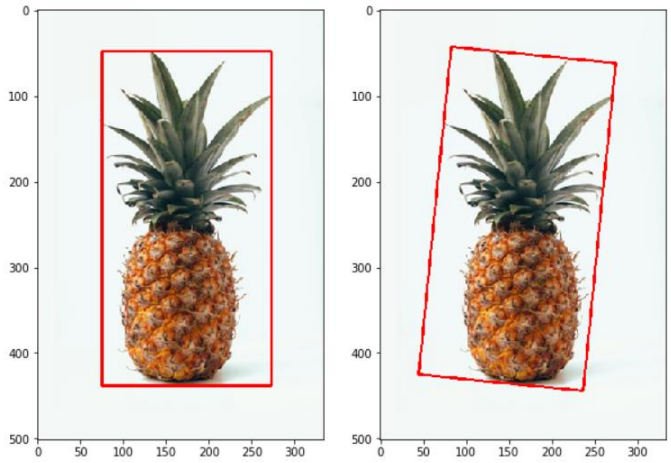
```
# The first order of the contours
c_0 = contours[0]# Get the 4 points of the bounding rectangle
x, y, w, h = cv2.boundingRect(c_0)# Draw a straight rectangle with the points
img_copy = img.copy()
img_box = cv2.rectangle(img_copy, (x, y), (x+w, y+h), color = (255, 0, 0), thickness = 2)
```

Note that this straight rectangle isn't the minimum among other possible boundings. We can extract the rectangle with the minimum area with the function `cv2.minAreaRect()` which finds a rotated rectangle enclosing the input 2D point set. After that, we get the 4 corners of this rectangle and put them at the `contour` parameter as shown below.

```
# Get the 4 points of the bounding rectangle with the minimum area
rect = cv2.minAreaRect(c_0)
box = cv2.boxPoints(rect)
box = box.astype('int')# Draw a contour with the points
img_copy = img.copy()
img_box_2 = cv2.drawContours(img_copy, contours = [box],
                             contourIdx = -1,
                             color = (255, 0, 0), thickness = 2)
```

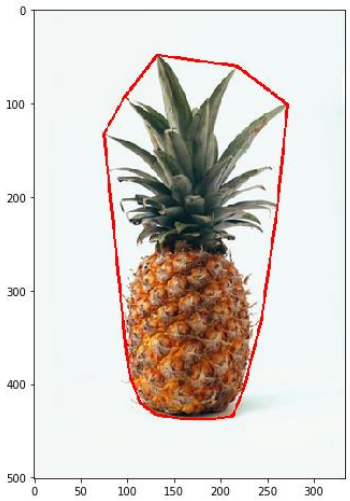
Now let's check the result and compare the two different contour boxes.

```
plt.figure(figsize = (10, 16))
plt.subplot(1, 2, 1); plt.imshow(img_box)
plt.subplot(1, 2, 2); plt.imshow(img_box_2)
```



We can also draw a convex shape contour with the function `cv2.convexHull()`. This takes a set of points and returns the convex hull from the given set. And by inputting this returned points as `contours` in `cv2.drawContours()` , we can get the convex contour as follows.

```
# Detect the convex contour
hull = cv2.convexHull(c_0)img_copy = img.copy()
img_hull = cv2.drawContours(img_copy, contours = [hull],
                            contourIdx = 0,
                            color = (255, 0, 0), thickness = 2)
plt.imshow(img_hull)
```

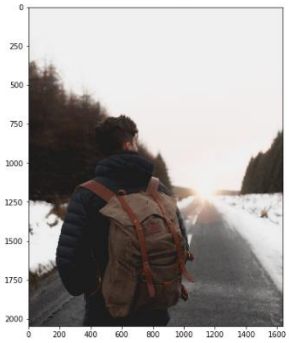


There are so many things to cover in contour detections. I encourage you to check **the documentation** and explore more options by yourself.

A little bit of fun with masking

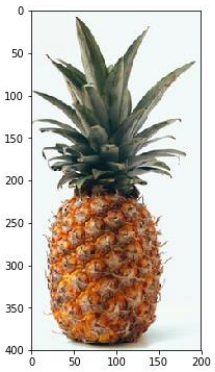
For the last exercise of this series, I'd like to have some fun time with doing image masking. We can add two different images with arithmetic operations such as image addition or bitwise operation. So our final task is attaching our Mr.pineapple on a man's left shoulder.

```
# Import the large image
backpacker = cv2.imread('images/backpacker.jpg')
backpacker = cv2.cvtColor(backpacker, cv2.COLOR_BGR2RGB)
plt.imshow(backpacker)
```



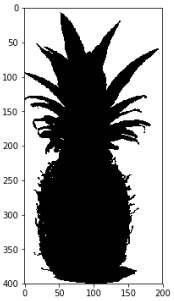
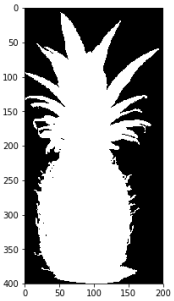
We'll cut the region of interest (the shoulder part) from the large image and the small image as shown below. Note that the size of the two images (the height and the width of the image) should be the same here.

```
# Crop the small image and the roi
roi = backpacker[750:1150, 300:500]
img_2 = img[40:440, 80:280]plt.figure(figsize = (6, 6))
plt.subplot(1, 3, 1); plt.imshow(roi)
plt.subplot(1, 3, 3); plt.imshow(img_2)
```

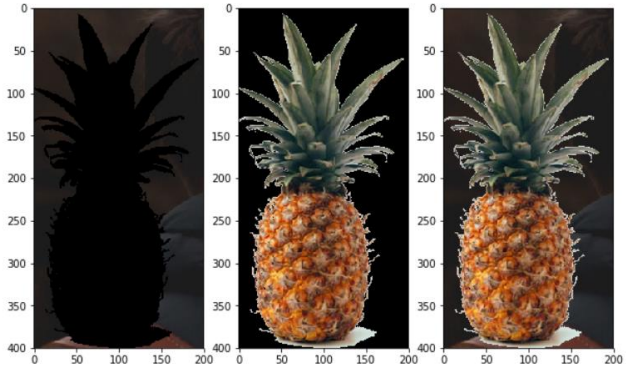
The next step is making the mask for each image. We're going to implement bitwise operation with the masks which should be a binary image. By thresholding the small image `img_2` , we create the mask. And then with `cv2.bitwise_not()`, we make another mask which is exactly the opposite of the first one.

```
# Creating the mask for the roi and small image
img_gray = cv2.cvtColor(img_2, cv2.COLOR_RGB2GRAY)
_, mask = cv2.threshold(img_gray, 254/2+100, 255, cv2.THRESH_BINARY)
mask_inv = cv2.bitwise_not(mask)plt.figure(figsize = (6, 6))
plt.subplot(1, 3, 1); plt.imshow(mask, cmap = 'gray')
plt.subplot(1, 3, 3); plt.imshow(mask_inv, cmap = 'gray')
```

If we implement the function `cv2.bitwise_and()` and the masks, they will pass only the white area of the image. Therefore if we apply the first mask to the `roi` image, we can set the background image from it. In the same way, if we apply the second one to the `img_2` image, we can make the foreground image with the fruit.

```
# Masking
img_bg = cv2.bitwise_and(roi, roi, mask = mask)
img_fg = cv2.bitwise_and(img_2, img_2, mask = mask_inv)
dst = cv2.add(img_fg, img_bg)plt.figure(figsize = (10, 6))
plt.subplot(1, 3, 1); plt.imshow(img_bg)
plt.subplot(1, 3, 2); plt.imshow(img_fg)
plt.subplot(1, 3, 3); plt.imshow(dst)
```



The outcome is a bit shambles considering the shadow at the bottom but let's keep it. If you'd like to learn how the bitwise operation works, a detailed explanation can be found **here**. Now we're ready to attach this combined

image to the original one. So we can simply do this by putting the `dst` image at the `roi` position as shown below.

```
# Final output
backpacker[750:1150, 300:500] = dst
display(backpacker)
```