5. Image Processing

To recap the previous discussions, a digital image is represented as an array of pixel values. Each pixel in the array represents a tiny element of the image, and its value determines the color or intensity of that element. The pixel values are represented as integers in the range of 0 to 255 for grayscale images, where 0 represents black and 255 represents white. For color images, pixel values are represented as a combination of blue, green, and red (BGR) channels, where each channel is in grayscale with a value between 0 and 255.

Digital images are represented by multi-dimensional arrays in OpenCV, a grayscale image is represented by a 2D array, and a color one is by a 3D array.

Image processing is a technique to apply some mathematical algorithms and use various functions and techniques to manipulate digital images in order to get the desired outcomes with enhanced effects, such as improving their quality or extracting useful information from them.

OpenCV provides a number of methods for image processing, this chapter will introduce some commonly and widely used methods for image processing.

This chapter begins with the color space conversion in section 5.1 which is the basics of image processing, we will introduce the conversion between grayscale, BGR and HSV color spaces.

Section 5.2 will explain how to resize, crop and rotate the images. This chapter will also introduce the object-oriented features of Python, we will create classes to

include the functions and write codes to instantiate the classes and invoke those functions.

Section 5.3 will explain how to adjust the Brightness and Contrast, and section 5.4 to adjust Hue, Saturation and Value of the images. By adjusting these values, the color and brightness of the images can be changed accordingly, this is a very important part of image processing.

Section 5.5 will introduce image blending which is to mix two images based on an algorithm. And Section 5.6 will introduce the bitwise operation and blending or mixing two images based on the bitwise operation, which can achieve a different effect.

Section 5.7 will describe image warping, a method to transform a skewed and distorted image into a straightened one.

Section 5.8 will introduce image blurring, it's a technique to make the image smooth by removing the noises from the image. There are many different ways to do it, this section will focus on two types of image blurring methods, Gaussian Blur and Median Blur.

Lastly, section 5.9 will introduce the histogram which gives us important information about the color distributions of the images.

Enjoy the Image Processing with OpenCV and Python.

5.1. Conversion of Color Spaces

```
Source:          ColorSpace.py
```

Section 3.4 has explained that the color image can be represented using either BGR or HSV color spaces, and a color image can also be converted to a grayscale image. OpenCV provides many methods for the conversion of color spaces, here we introduce two of them: BGR to and from Grayscale, and BGR to and from HSV.

5.1.1. Convert BGR to Gray

There are two ways to obtain a grayscale image from the color one, 1) use `cv2.cvtColor()` function with `cv2.COLOR_BGR2GRAY` as its second parameter, and 2) load the image with grayscale mode by using `cv2.imread()` function with `cv2.IMREAD_GRAYSCALE` as the second parameter.

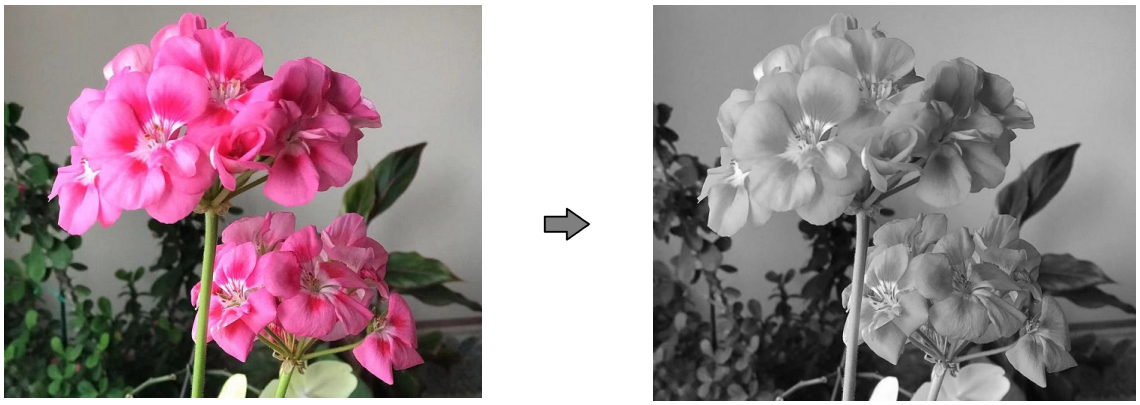
The first method loads the color image and then converts it to grayscale, then we have both the color and grayscale images available in the memory.

However, if the color image is not needed, the second method loads the image in grayscale only, the color one is not available in this case and could save some memory. Below code snippets show the two methods:

```
1  def convert_bgr2gray(image):
2      gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
3      return gray
4
5  def load_image_gray(file_name):
6      gray = cv2.imread(file_name,
                         cv2.IMREAD_GRAYSCALE)
7      return gray
```

Line 1 to 3 is the first method, the color image in `image` is passed as a parameter, `cv2.cvtColor()` function will convert it to grayscale one.

Line 5 to 7 is the second method, it loads the image in grayscale mode directly from the file.



Color Image *Grayscale Image*

Figure 5.1 Convert BGR to Grayscale

As shown in Figure 5.1 above, the color image (left) is converted to a grayscale one (right).

As discussed earlier, in OpenCV the default color space for an image is BGR, so a color image can be split into three channels of blue, green and red, each channel is in grayscale, as shown in Figure 5.2.

Figure 5.2 Color Image is Made of Blue, Green and Red Channels

The `cv2.split()` function can be used to split a color image into the three channels:

```
1  def split_image(image):
2      ch1, ch2, ch3 = cv2.split(image)
3      return ch1, ch2, ch3
```

And each channel is in grayscale.

5.1.2. Convert Grayscale to BGR

A grayscale image can also be converted to BGR color space, in this case the source is a grayscale image which only has one channel and does not have any color information, therefore the result BGR image looks the same as the original grayscale one. But the difference is the BGR image has three channels although looks the same as the grayscale one, while the original gray image only has one channel.

To convert a grayscale image to BGR, use `cv2.cvtColor()` function with `cv2.COLOR_GRAY2BGR` as the second parameter,

```
1  def convert_gray2bgr(image):
2      bgr = cv2.cvtColor(image,cv2.COLOR_GRAY2BGR)
3      return bgr
```

Figure 5.3 below shows the idea:

Grayscale (one channel) *BGR (three channels)*

Figure 5.3 Convert Grayscale to BGR

The left image is the original grayscale image, which has only one channel. The right image, however, is in BGR color space although looks same as the grayscale one, it can be split into three channels.

Since the right image looks same as the left one, why we should convert it? There are several reasons to do it, sometimes in the image processing, some operations need to be performed on two or more images, it's important that all the images are in the same color space. For example, if we want to blend, or mix, two images together, but one is color image and another is grayscale, the grayscale one must be converted into BGR before blending them.

Sometimes when you want to colorize a grayscale image, it must be converted to BGR first, then paint colors on different channels, because you are not able to paint colors on a single-channel grayscale image.

5.1.3. Convert BGR to HSV

As explained in section 3.4, an image can be represented not only in BGR but also in HSV color spaces, OpenCV can easily convert the image between BGR and HSV. Like converting it to grayscale, the same function `cv2.cvtColor()` is used but the second parameter is different, `cv2.COLOR_BGR2HSV` in this case.

The HSV image can also be split into three channels, hue, saturation and value. Then each channel can be adjusted to achieve some different effects, later we will explain how to adjust each channel to change the image.

Below code snippets are to convert an image to HSV color space:

```
1  def convert_bgr2hsv(image):
2      hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
3      return hsv
```

As shown in Figure 5.4, the color image is converted to HSV color space, and then it can be further split into three channels in hue, saturation and value by using `cv2.split()` function.

Figure 5.4 Convert BGR to HSV

5.1.4. Convert HSV to BGR

When OpenCV displays an image, the function `cv2.imshow()` is used and by default in BGR color space. If an image is converted to HSV and followed by some image processing operations, it can not be directly sent to `cv2.imshow()` for displaying, instead it must be converted back to BGR, then sent to `cv2.imshow()` for displaying, otherwise the image will not be displayed correctly.

In last section an image is converted to HSV, and here we will convert it back to BGR with the same function `cv2.cvtColor()` but different parameter `cv2.COLOR_HSV2BGR`.

```
1  def convert_hsv2bgr(image):
2      bgr = cv2.cvtColor(image, cv2.COLOR_HSV2BGR)
3      return bgr
```

The example in section 5.4 later will show how to convert a BGR image to HSV, and make some adjustments to hue, saturation and value to achieve some effects, and then convert it back to BGR for displaying.

5.2. Resize, Crop and Rotate an Image

```
Source:          ResizeCropRotate.py

Library:         common/ImageProcessing.py
```

Python is an excellent language to support object-oriented programming, this chapter will start to use object-oriented techniques to write the codes. We will create classes to include properties and functions, then instantiate the classes and invoke their functions.

Now create a class called `ImageProcessing`, and define three functions inside, `resize()`, `rotate()` and `crop()`. When a class is created it always has a built-in `__init__()` function, which is always executed when the class is instantiated. It's used to assign initial values to the properties or other operations that are necessary when the object is created. Here we set the `window_name` and `image_name` properties inside `__init__()` function.

The full source code is in the *ImageProcessing.py* in the common folder of the Github repository.

```
1   import cv2
2
3   class ImageProcessing(object):
```

```
4       def __init__(self, window_name, image_name):
5           self.window_name = window_name
6           self.image_name = image_name
7           self.image = cv2.imread(self.image_name)
8
9       def show(self, title=None, image=None):
10          if image is None:
11              image = self.image
12          if title is None:
13              title = self.window_name
14          cv2.imshow(title, image)
```

Explanations:

Line 3	*Define the `ImageProcessing` class*
Line 4	*The built-in `__init__()` function, it's always executed when the class is instantiated. The parameters should be passed when instantiating the class.*
Line 5 - 7	*Set the class properties, `window_name` and `image_name`, then load the image using `cv2.imread()`.*
Line 9 - 14	*Define `show()` function, which will show the image using `cv2.imshow()`. When the parameters specify the image or window name then show that image with window name, otherwise use the image and window name specified at `__init()__` function when the class is initialized.*

To continue, add three functions, `resize()`, `crop()` and `rotate()`, to the class:

```
16      def resize(self, percent, image=None):
17          if image is None:
18              image = self.image
19          width = int(image.shape[1]*percent/100)
```

```
20          height = int(image.shape[0]*percent/100)
21          resized_image = cv2.resize(image,
                                       (width, height) )
22          return resized_image
23
24      def crop(self,pt_first,pt_second,image=None):
25          if image is None:
26              image = self.image
27          # top-left point
            x_tl, y_tl = pt_first
28          # bottom-right point
            x_br, y_br = pt_second
29          # swap x value if opposite
            if x_br < x_tl:
30              x_br, x_tl = x_tl, x_br
31          # swap y value if opposite
            if y_br < y_tl:
32              y_br, y_tl = y_tl, y_br
33          cropped_image=image[y_tl:y_br,x_tl:x_br]
34          return cropped_image
35
36      def rotate(self,angle,image=None,scale=1.0):
37          if image is None:
38              image = self.image
39          (h, w) = image.shape[:2]
40          center = (w / 2, h / 2)
41          rot_mat = cv2.getRotationMatrix2D(center,
                                      angle, scale)
42          rotated_image = cv2.warpAffine(image,
```

```
                                    rot_mat, (w, h))
43          return rotated_image
```

Explanations:

Line 16	*Define `resize()` function, specify percentage as parameter, and optionally the image as parameter.*
Line 17 - 18	*If `image` is not specified in the parameters, then use the class property's image.*
Line 19 - 22	*Calculate the resized width and height based on the percentage. Then call `cv2.resize()` function to resize the image.*
Line 24 - 34	*Define `crop()` function, similar to what we did in Section 4.4.*
Line 36 – 44	*Define `rotate()` function, pass `angle` as a parameter and optionally `image` and `scale`. Use `cv2.getRotationMatrix2D()` and `cv2.warpAffine()` functions to perform the rotation.*

Reference the OpenCV documents for `getRotationMatrix2D()` and `warpAffine()` functions at the below link:

https://docs.opencv.org/4.7.0/da/d54/group__imgproc__transform.html

The class definition for `ImageProcessing` is done for now, more functions will be added later.

Now the class can perform the resize, crop and rotate operations.

In the source codes of *ResizeCropRotate.py* in the Github repository, the class is imported in Line 2, this is a typical way to include another Python file for reference.

```
1   import cv2
2   import common.ImageProcessing as ip
3
4   if __name__ == "__main__":
5       # Create an ImageProcessing object
6       ip = ip.ImageProcessing("Resize,Crop and Rotate",
```

```
                                    "../res/flower005.jpg")
7
8       # Show original image
9       ip.show()
10
11      # Resize the original image and show it
12      resized_image = ip.resize(50)
13      ip.show("Resized -- 50%", resized_image)
14
15      # Rotate the resized image and show it
16      rotated_image = ip.rotate(45, resized_image)
17      ip.show("Rotated -- 45 degree", rotated_image)
18
19      # Crop the original image and show it
20      cropped_image = ip.crop((300, 10), (600, 310))
21      ip.show("Cropped", cropped_image)
22
23      cv2.waitKey(0)
24      cv2.destroyAllWindows()
```

Explanations:

Line 2	*Import the class file we defined earlier, because it is in* `common` *folder,* `common.file_name` *is used to locate the file.*
Line 4	*The main entrance, when the code is executed, it starts from here.*
Line 6	*Instantiate* `ImageProcessing` *class with parameters, the window's name and the path of an image, which are set as properties of this class.*
Line 9	*Show the default image and default window name, which are specified at the class instantiation.*

Line 12 – 13	*Invoke the `resize()` function to resize the image, and show it.*
Line 16 – 17	*Invoke the `rotate()` function and show the result with 45-degree rotated.*
Line 20 – 21	*Invoke the `crop()` function and pass two points to crop the image, and show it.*

Below are the results, Figure 5.5 shows the original image:

Figure 5.5 Original Image

Figure 5.6 shows the 50% resized image:

Figure 5.6 50% Resized Image

Figure 5.7 shows the cropped image:

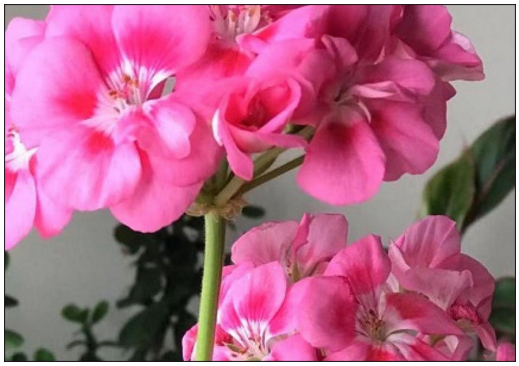


Figure 5.7 Cropped Image

And finally, Figure 5.8 shows the 45-degree rotated image:

Figure 5.8 45 Degree Rotated Image

Feel free to play with the source codes, for example change an image file, change the degree of rotation, change the percentage of the resizing, and observe how the image is processed.

5.3. Adjust Contrast and Brightness of an Image

```
Source:          ContrastBrightness.py

Library:         common/ImageProcessing.py
```

Section 4.5 has an example to adjust the brightness and contrast for a webcam, however unfortunately OpenCV doesn't provide functions to adjust the brightness and contrast for an image. But OpenCV official document recommends an alternative way to do it, see the link at:

https://docs.opencv.org/4.7.0/d3/dc1/tutorial_basic_linear_transform.html

This book is not intended to deep dive into the algorithms or theories, instead, we focus on the implementation of the techniques using OpenCV with Python. However, in order to make things clear, sometimes we will summarize, or quote from other sources, the related algorithms or theories.

To summarize the techniques of contrast and brightness adjustment, the below formula from OpenCV official document depicts the calculation of the contrast and brightness,

$$g(i,j) = \alpha\, f(i,j) + \beta$$

from OpenCV official document at
https://docs.opencv.org/4.7.0/d3/dc1/tutorial_basic_linear_transform.html

$f(i,j)$ is the original image and $g(i,j)$ is the result. α changes the contrast, greater than 1 for higher contrast, less than 1 for lower contrast. It should be greater than 0. β changes the brightness, values vary from -127 to +127. (i,j) indicates the coordinates of the image pixels.

The OpenCV official document provides an example to explain how to use the Python array and matrix operations to implement the above formula.

Alternatively, OpenCV provides `cv2.addWeighted()` function which is designed for blending two images with weight added on each, we will use this function to implement the above formula to adjust the contrast and brightness.

```
cv2.addWeighted(image, contrast, zeros, 0, brightness)
```

The function `cv2.addWeighted()` implements the below formula:

$$g(i,j) = \alpha_1 f_1(i,j) + \alpha_2 f_2(i,j) + \beta$$

$g(i, j)$ is the result, $f1(i, j)$ is the image and passed to `cv2.addWeighted()` as the first parameter, α_1 is the second parameter which is contrast; because only one image is interested in this example, so $f2(i, j)$ is an all-zero image, and α_2 is also a zero, therefore the third and fourth parameters of the function are an all-zero array and a zero value. β is the last parameter, which is brightness, this value will be added to the result.

Now add a new `contrast_brightness()` function to the class `ImageProcessing`, which we have defined in section 5.2.

```
45  def contrast_brightness(self, contrast,
                            brightness, image=None):
46      # contrast:
        # between 0 and 1: less contrast;
47      #          > 1: more contrast;
48      #          1: unchanged
49      # brightness: -127 to 127;
50      #          0 unchanged
51      if image is None:
52          image = self.image
53      zeros = np.zeros(image.shape, image.dtype)
54      result = cv2.addWeighted(image, contrast,
                                 zeros, 0, brightness)
55      return result
```

Explanations:

Line 45	*Define `contrast_brightness()` function, specify contrast and brightness as parameters, and optionally an image.*
Line 53	*Create an all-zero array, because `cv2.addWeighted()` requires two images, we don't use the second one, so pass the all-zero array to it.*
Line 54	*Call `cv2.addWeighted()` function to apply the weights to the image.*

Below is the code to adjust the brightness and contrast using the function just created above. Like Section 4.5, two trackbars are added for users to adjust the brightness and contrast.

```
1   import cv2
2   import common.ImageProcessing as ip
3
4   def change_brightness(value):
5       global brightness
6       brightness = value - 128
7
8   def change_contrast(value):
9       global contrast
10      contrast = float(value)/100
11
12  if __name__ == "__main__":
13      brightness = 0
14      contrast = 1.0
15      title = "Adjust Brightness and Contrast"
16      ip = ip.ImageProcessing(title,
                                "../res/flower003.jpg")
17      cv2.createTrackbar('Brightness', title, 128,
                                255, change_brightness)
18      cv2.createTrackbar('Contrast', title, 100,
                                300, change_contrast)
19
20      while True:
21          adjusted_image = ip.contrast_brightness(
                                contrast, brightness)
```

```
22            ip.show(image=adjusted_image)
23            if cv2.waitKey(10) & 0xFF == 27:
24                break
25        cv2.destroyAllWindows()
```

Explanations:

Line 2	*Import the `ImageProcessing` class.*
Line 4 - 6	*Define the callback function for the trackbar to adjust the brightness.*
Line 8 - 10	*Define the callback function for the trackbar to adjust the contrast.*
Line 12	*Main entrance.*
Line 16	*Instantiate the `ImageProcessing` object.*
Line 17 - 18	*Create two trackbars for brightness and contrast.*
Line 21 – 22	*Call `contrast_brightness()`, the function we just defined, to adjust brightness and contrast.*

Execute the code, the result shows the image together with two trackbars, as Figure 5.9, one is for adjusting brightness and another for contrast:

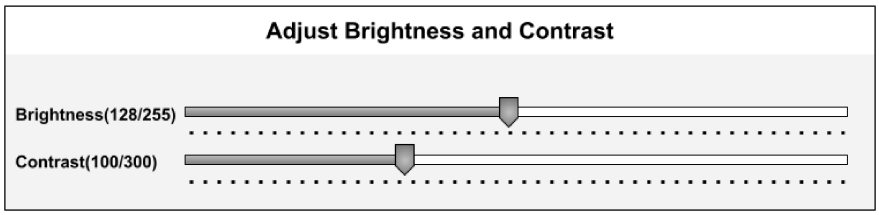


Figure 5.9 Trackbars to Adjust Brightness and Contrast

Use the mouse to drag the two trackbars to change the brightness and contrast values, the image will be changed accordingly in real-time.

The range of contrast in trackbar is from 0 to 300, and the default is 100. In the callback function `change_contrast()`, the value is divided by 100, therefore the contrast range is from 0.0 to 3.0, default is 1.0.

Feel free to play with the codes by changing the image files and adjusting the two trackbars and observe how the image is changing accordingly.

5.4. Adjust Hue, Saturation and Value

```
Source:          HueSaturation.py

Library:         common/ImageProcessing.py
```

Section 3.4.3 has explained a color image can be split not only to BGR channels but also HSV channels, the former represents blue, green and red channels, the latter represents hue, saturation and value. The hue represents the color of the image, different hue values have different colors; saturation represents the purity of the color, a higher saturation means a more colorful image; while value represents the brightness of the image, a higher value means a brighter image.

In this section we will explain how to adjust hue, saturation and value of an image, three trackbars will be created for the three variables, we will see how the image is changing when users adjust these variables in real-time.

The first thing is to convert the image into HSV using `cv2.cvtColor()` with `cv2.COLOR_BGR2HSV` parameter, then split it into hue, saturation and value channels using `cv2.split()`.

Then add weights to hue, saturation and value channels respectively, and merge them back to HSV. Finally convert the HSV to BGR for displaying.

Same as the previous section, `cv2.addWeighted()` is used to adjust each channel with a weight. The below line of code is to add a weight `h_weight` to the `hue` channel,

```
1  hue = cv2.addWeighted(hue, 1.0, zeros, 0, h_weight)
```

And add the weights to other channels in the same way.

Here are the codes, the function `hue_saturation_value()` is added into our `ImageProcessing` class:

```
1   def hue_saturation_value(self, hue, saturation,
```

```
                                value, image=None):
2       if image is None:
3           image = self.image
4       hsvImage = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
5       h, s, v = cv2.split(hsvImage)
6       zeros = np.zeros(h.shape, h.dtype)
7       h = cv2.addWeighted(h, 1.0, zeros, 0, hue)
8       s = cv2.addWeighted(s, 1.0, zeros, 0, saturation)
9       v = cv2.addWeighted(v, 1.0, zeros, 0, value)
10      result = cv2.merge([h, s, v])
11      result = cv2.cvtColor(result, cv2.COLOR_HSV2BGR)
12      return result
```

Explanations:

Line 1	*Define the function in* `ImageProcessing` *class, the parameters are* `hue`, `saturation` *and* `value`.
Line 4	*Convert the image into HSV image.*
Line 5	*Split the HSV image into separate* `h`, `s`, `v` *channels.*
Line 7 - 9	*Add the weights to the three channels. The* `hue`, `saturation` *and* `value` *are the weights to* `h`, `s`, *and* `v` *channels respectively.*
Line 10	*Merge the* `h`, `s`, `v` *channels into the HSV image.*
Line 11	*Convert the HSV image back to BGR image.*

This function is called from the *HueSaturation.py* file, in this example we add a feature to save the image into a file on the local disk drive, in addition to pressing ESC key to exit, when press "s" key the file will be saved to a specified file using `cv2.imwrite()` function.

```
1   import cv2
2   import common.ImageProcessing as ip
3
```

```
4   def change_hue(value):
5       global hue
6       hue = (value * 2) - 255
7
8   def change_saturation(value):
9       global sat
10      sat = (value * 2) - 255
11
12  def change_value(value):
13      global val
14      val = (value * 2) - 255
15
16  if __name__ == "__main__":
17      hue, sat, val = 0, 0, 0
18      title = "Adjust Hue, Saturation and Value"
19      ip = ip.ImageProcessing(title,
                        "../res/flower003.jpg")
20      cv2.createTrackbar('Hue', title, 127, 255,
                        change_hue)
21      cv2.createTrackbar('Saturation', title,
                    127, 255, change_saturation)
22      cv2.createTrackbar('Value', title, 127,
                    255, change_value)
23      ip.show("Original")
24      print("Press s key to save image,
              ESC to exit.")
25
26      while True:
27          adjusted_image =
              ip.hue_saturation_value(hue, sat, val)
```

```
28            ip.show(image=adjusted_image)
29            ch = cv2.waitKey(10)
30            if (ch & 0xFF) == 27:
31                Break
32            elif ch == ord('s'):
33                # press 's' key to save image
34                filepath = "C:/temp/flower003.png"
35                cv2.imwrite(filepath, adjusted_image)
36                print("File saved to " + filepath)
37        cv2.destroyAllWindows()
```

Explanations:

Line 4 - 15	*Define three callback functions for trackbars for hue, saturation and value. The value is changing from -255 to 255 for each. You can modify the range base on your needs.*
Line 19	*Instantiate the `ImageProcess` class with a title and an image.*
Line 20 - 22	*Create three trackbars for hue, saturation and value respectively.*
Line 23	*Show the original image.*
Line 27	*Call the function we just defined in `ImageProcess` class with parameters of the adjusted hue, saturation and value.*
Line 28	*Show the result image.*
Line 32 - 36	*Save the result image when pressing "s" key.*

Execute the code, the result shows the original image and the adjusted image together with three trackbars, as Figure 5.10, for adjusting hue, saturation and value respectively.

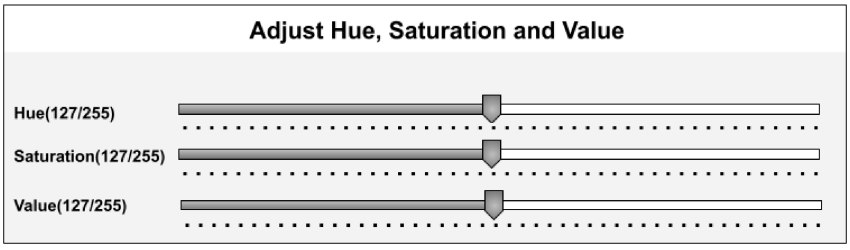


Figure 5.10 Trackbars to Adjust Hue, Saturation and Value

The original image and adjusted image are omitted here, feel free to change different images in the codes and observe how they are changing when adjusting the hue, saturation and value by moving the knobs of the trackbars.

5.5. Blend Image

```
Source:          BlendImage.py

Library:         common/ImageProcessing.py
```

Blending images refers to the process of combining two images to create a single image that has the characteristics of both original images. The result is a combination of the corresponding pixel values of the two original images with weights.

The blending process involves specifying a weight for each pixel in one of the original images, computing (1 – weight) for each pixel in another image, and then merging them together to produce the output image.

Blending images is a common operation in computer vision and image processing, and is used for a variety of applications, such as creating panoramas, compositing images, and generating special effects in videos and images.

Say, there are two images, Original Image 1 and Original Image2, as shown in Figure 5.11 and Figure 5.12.

Figure 5.11 Blend Image: Original Image 1

Figure 5.12 Blend Image: Original Image 2

The OpenCV documents describe the algorithm to blend them together,

$$g(i,j) = \alpha f_0(i,j) + (1-\alpha) f_1(i,j)$$

from OpenCV official document at
https://docs.opencv.org/4.7.0/d5/dc4/tutorial_adding_images.html

$f_0(i,j)$ and $f_1(i,j)$ are the two original images, and $g(i,j)$ is the output image. α is the weight (value from 0 to 1) on the first image, $(1-\alpha)$ is the weight on the second image.

By adjusting the value of α, we can achieve different blending effects. Same as the previous sections, `cv2.addWeighted()` function is used to blend the two images.

The `blend()` function is to be added to the `ImageProcessing` class:

```
1       def blend(self, blend, alpha, image=None):
2           if image is None:
3               image = self.image
4           blend = cv2.resize(blend,
                    (image.shape[1], image.shape[0]))
5           result = cv2.addWeighted(image, alpha,
                         blend, (1.0 - alpha), 0)
6           return result
```

Explanations:

Line 1	*Define `blend()` function, the parameters are the image to blend, and alpha which is between 0 and 1.*
Line 4	*The two images must be in the same size, make them the same using `cv2.resize()`.*
Line 5	*Use `cv2.addWeighted()` function to implement the blending algorithm.*

Here are the main codes:

```
1   def change_alpha(value):
2     global alpha
3     alpha = float(value)/100
4   def blendTwoImages(imageFile1, imageFile2):
5     global alpha
6     alpha = 0.5
7     title = "Blend Two Images"
8     iproc = ip.ImageProcessing(title, imageFile1)
9     toBlend = cv2.imread(imageFile2)
10    cv2.createTrackbar("Alpha",title,50,100,change_alpha)
11    iproc.show("Original Image 1", )
12    iproc.show("Original Image 2", toBlend)
13    print("Press s key to save image, ESC to exit.")
14    while True:
15      blended_image = iproc.blend(toBlend, alpha)
```

```
16      iproc.show(image=blended_image)
17      ch = cv2.waitKey(10)
18      if (ch & 0xFF) == 27:
19        break
20      elif ch == ord('s'):
21        # press 's' key to save image
22        filepath = "C:/temp/blend_image.png"
23        cv2.imwrite(filepath, blended_image)
24        print("File saved to " + filepath)
25    cv2.destroyAllWindows()
26
27  if __name__ == "__main__":
28    image1 = "../res/sky001.jpg"
29    image2 = "../res/bird002.jpg"
30    blendTwoImages(image1, image2)
```

A trackbar is created for adjusting `alpha` from 0 to 100, with a default value of 50. The callback function `change_alpha()` divides the value by 100, then it's from 0 to 1.0.

The result looks something like Figure 5.13:

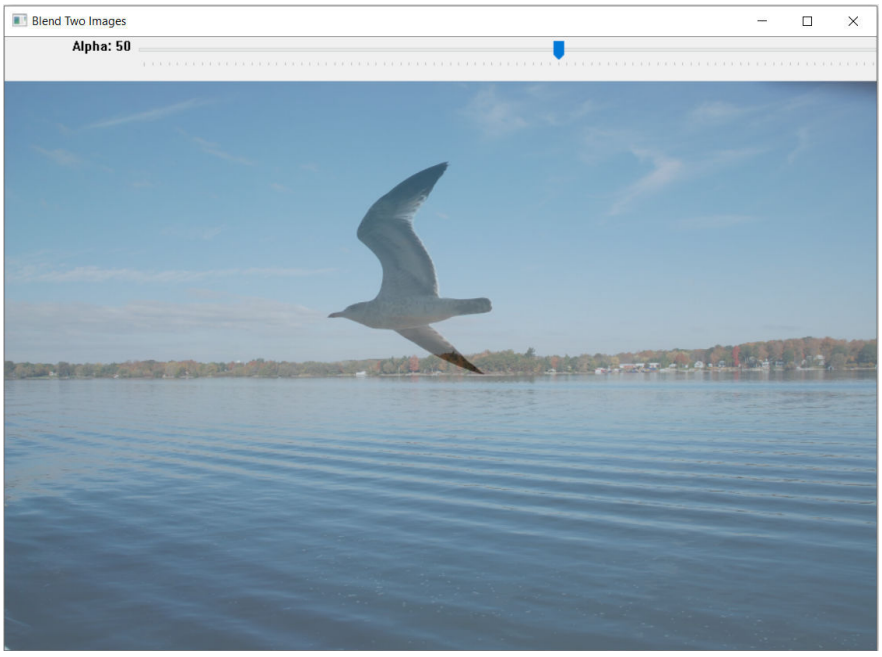


Figure 5.13 Blend Images

The default `alpha` of 50 makes the two original images blend evenly. By adjusting the `alpha` value, the output image will be changing. If `alpha=100`, the result becomes pure Original Image 1; if `alpha=0`, the result is pure Original Image 2. If the `alpha` is in between, the result is mixed, but the weight of each image is different based on the `alpha` value.

5.6. Bitwise Operation

```
Source:          BlendImageBitwise.py

Library:         common/ImageProcessing.py
```

The last section introduced image blending by implementing the algorithm introduced by the OpenCV document. As you can see from the result in Figure 5.13, the image looks a little bit faded out, and both original images become transparent to some extent. It depends on what effects you want to achieve, sometimes this kind of faded-out effect is not ideal, you might want the images to be opaquely added together without fading out.

This section will introduce another way to blend two images, a blending mask will be used in the image blending. The blending mask is a grayscale image that specifies the contribution of each pixel in the input images to the output image. It is also referred to as an alpha mask or alpha matte.

The blending mask is used as a weighting function to compute the weighted average of the pixel values from the input images. The values of the blending mask typically range from 0 to 255, where 0 represents complete opacity and 255 represents complete transparency. The blending mask is usually a grayscale image, but it can also be a color image in some cases. However, in this section we use a binary mask, the values are either 0 or 255, with no values in between.

In this section we will create a mask from the image of Figure 5.12, the bird image, and apply it to both images as a bitwise operation to achieve a different blending effect.

OpenCV provides bitwise operations – AND, OR, NOT and XOR. It is very useful when we want to extract something partially from an image and put it in another image. These bitwise operations can be used here to achieve the effects we want.

`cv2.bitwise_and(img1,img2)`	Perform bitwise AND for img1 and img2
`cv2.bitwise_or(img1,img2)`	Perform bitwise OR for img1 and img2
`cv2.bitwise_not(img1)`	Perform bitwise NOT for img1
`cv2.bitwise_xor(img1,img2)`	Perform bitwise XOR for img1 and img2

The algorithm is as following:

$$g(i,j) = mask \,\&\, f_0(i,j) + (1 - mask) \,\&\, f_1(i,j)$$

$f_0(i,j)$ and $f_1(i,j)$ are the original images to blend, and $g(i,j)$ is the result image; $\&$ is the bitwise AND operation.

The bitwise blending process is shown in Figure 5.14, a $mask$ is created based on Original Image 2 which is Figure 5.12. And $(1 - mask)$ is created based on the $mask$.

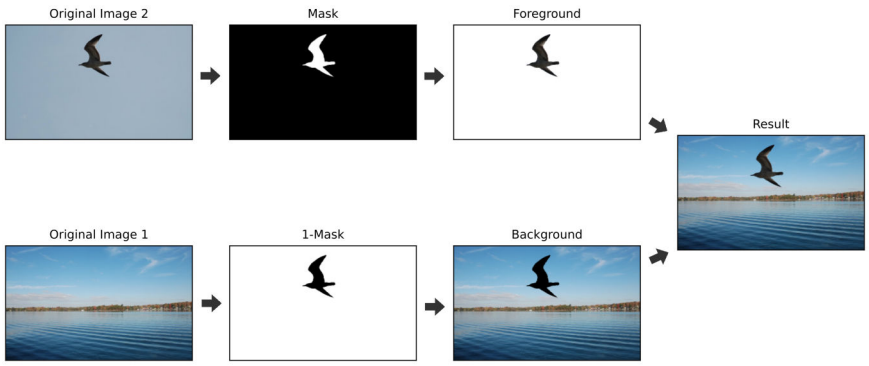


Figure 5.14 Blending Images with Bitwise

The $mask$ is applied to Original Image 2 by bitwise AND, and produces the foreground of the output image, only the bird appears transparently in the foreground image, and other pixels are opaque and appears as black in the foreground, as shown in Figure 5.15.

Then $(1 - mask)$ is applied to the Original Image 1 by bitwise AND, and produces the background image. The bird appears black here, and all other pixels appear transparently.

Finally, the foreground and background are joined together by bitwise OR to produce the result, as shown in Figure 5.14.

There are different ways to create a mask based on an image. An effective way to create a mask is introduced in Section 6.8.1 later, the image is converted to HSV color space, and find out the lower and upper range of HSV value to pick up the interested part of the image, in our case the bird. Then use `cv2.inRange()` function to get the mask. And lastly convert the mask image from grayscale to BGR color space.

Below are the code snippets to create our mask. Note, the creation of masks depends on the images, different images have different methods to create masks, section 6.8 will explain it in detail. Below code snippets are using the technique introduced in Section 6.8.1,

```
1  def remove_background_by_color(
                self,hsv_lower,hsv_upper,image=None):
2    if image is None:
3      image = self.image
4    imgHSV = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
5    mask = cv2.inRange(imgHSV, hsv_lower, hsv_upper)
6    mask = 255 - mask
7    mask = cv2.cvtColor(mask, cv2.COLOR_GRAY2BGR)
8    bg_removed = cv2.bitwise_and(image, mask)
9    return bg_removed, mask
```

Figure 5.15 shows the foreground image by applying the $mask$ on the bird image with bitwise AND operation.

Figure 5.15 Foreground Image After $mask$ Applied

As shown in the above result, all the background from the original image is removed, only the bird is left, which is the interesting part of this image.

This implements the first part of the above formula:

$$g(i,j) = mask \,\&\, f_0(i,j) + (1 - mask) \,\&\, f_1(i,j)$$

Next is to create $(1 - mask)$, technically this is 255 – $mask$, because the mask is created in a grayscale channel, which has values ranging from 0 to 255. However, in this case we deal it with the binary channel, the values are either 0 or 255, meaning either 100% opacity or 100% transparency, no other values in between. Although in other cases the values in between could indicate the percentage of transparency.

Figure 5.16 shows the result of $(1 - mask)$,

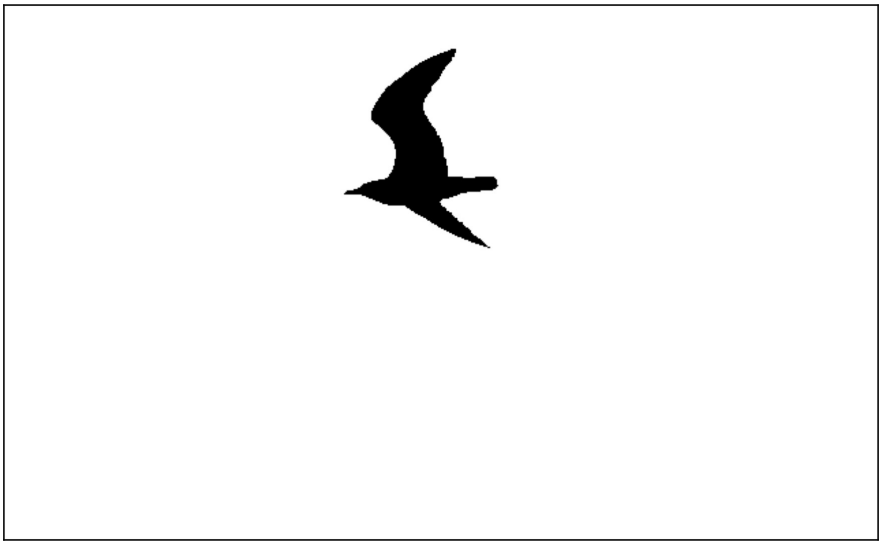


Figure 5.16 *(1 - mask)* Created based on *mask*

Then apply $(1 - mask)$ to the second image as bitwise AND operation to produce the background image, the result is shown in Figure 5.17.

Figure 5.17 Background Image After *(1-mask)* Applied

The pixels of the bird are removed, and all others remain on the result image. This implements the second part of the formula:

$$g(i,j) = mask \,\&\, f_0(i,j) + (1 - mask) \,\&\, f_1(i,j)$$

Finally, the two images are merged by bitwise OR operation, or just simply add them up, the result is shown as Figure 5.18:

Figure 5.18 Blended Image with Bitwise Operations

This implements the whole formula:

$$g(i,j) = mask \,\&\, f_0(i,j) + (1 - mask) \,\&\, f_1(i,j)$$

By comparing it with the one in the last section, the difference is obvious, this one does not have any faded-out effect.

In this case the foreground and background are merged with 100% transparency. In real-world projects, depending on what effects you want to achieve, different methods can be selected.

Below is the code snippet in the `ImageProcessing` class,

```
1  def blend_with_mask(self, blend, mask, image=None):
2    if image is None:
3      image = self.image
4    blend = cv2.resize(blend, image.shape[1::-1])
5    mask = cv2.resize(mask, image.shape[1::-1])
6    result = cv2.bitwise_and(blend, mask) +
               cv2.bitwise_and(image,(255-mask))
7    return result
```

Explanations:

Line 1	*Define `blend_with_mask()` function, the blend image and mask image are passed as parameters.*
Line 4 - 5	*The mask and blend images must be in the same size as the original image, use `cv2.resize()` to make them the same dimension, in case they are not.*
Line 6	*Use bitwise AND on blend image and mask image to produce the foreground image.* *Use bitwise AND again on the original image with (255-mask) to produce the background image.* *Then add both images using plus operation to make the resulting image. `cv2.bitwise_or()` can also be used, they are the same in this case.*

The full source codes for this example are in *BlendImageBitwise.py* file.

```
1   import cv2
2   import common.ImageProcessing as ip
3
4   def blendTwoImagesWithMask(imageFile1, imageFile2):
5       title = "Blend Two Images"
6       iproc = ip.ImageProcessing(title, imageFile1)
7       iproc.show(title="Original Image1",
                   image=iproc.image)
8       toBlend = cv2.imread(imageFile2)
9       iproc.show(title="Original Image2", image=toBlend)
10      _, mask = iproc.remove_background_by_color(
                        hsv_lower = (90, 0, 100),
11                      hsv_upper = (179,255,255),
12                      Image = toBlend   )
13      iproc.show(title="Mask from Original Image2",
                   image=mask )
14      iproc.show(title="(1-Mask)", image= (255-mask))
15      blend = iproc.blend_with_mask(toBlend, mask)
16      iproc.show(title=title, image=blend)
```

```
17        cv2.waitKey(0)
18        cv2.destroyAllWindows()
19
20    if __name__ == "__main__":
21        image1 = "../res/sky001.jpg"
22        image2 = "../res/bird002.jpg"
23        blendTwoImagesWithMask(image1, image2)
```

Line 10 – 12 is to call `remove_background_by_color()` function in `ImageProcessing` class to get the mask of the bird image, the details will be explained in Section 6.8.1. The value of parameters of `hsv_lower` and `hsv_upper` are specific only to this image, different images should have different values to get a mask.

5.7. Warp Image

```
Source:          WarpImage.py
Library:         common/ImageProcessing.py
                 common/Draw.py
```

Warp image refers to the process of geometrically transforming an image into different shapes. It involves applying a perspective or affine transformation to the image, which can change its size, orientation, and shape.

This section introduces *perspective warping*, also known as *perspective transformation*, which is a type of image warping that transforms an image from one perspective to another. It is a geometric transformation that changes the viewpoint of an image, as if the observer's viewpoint has moved or rotated in space.

It is often used to correct the distortion caused by the camera's perspective when capturing an image. This distortion causes objects in the image to appear different in size and shape depending on their position in the image. For example, objects closer to the camera appear larger and more distorted than those farther away.

The perspective warping process includes defining four points in the original image and mapping them to four corresponding points in the output image. These points are known as the *source points* and *destination points*, respectively. Once the corresponding points are identified, a transformation matrix is calculated, which maps each pixel in the original image to its new location in the output image. As shown in Figure 5.19, there is an original image in the left, and the output image in the right. The four source points from the original image are A, B, C and D, the perspective warping process will transform them to the output image in the right, so that A is mapped to A', B to B', C to C' and D to D'.

The real-world use case is, for example, using a camera to take a picture of a document, there are some distortions that make the document looks like the left-side image where A, B, C and D are the four corners of the document. A perspective warping will be able to transform the document into the right-side image, which is corrected and aligned properly.

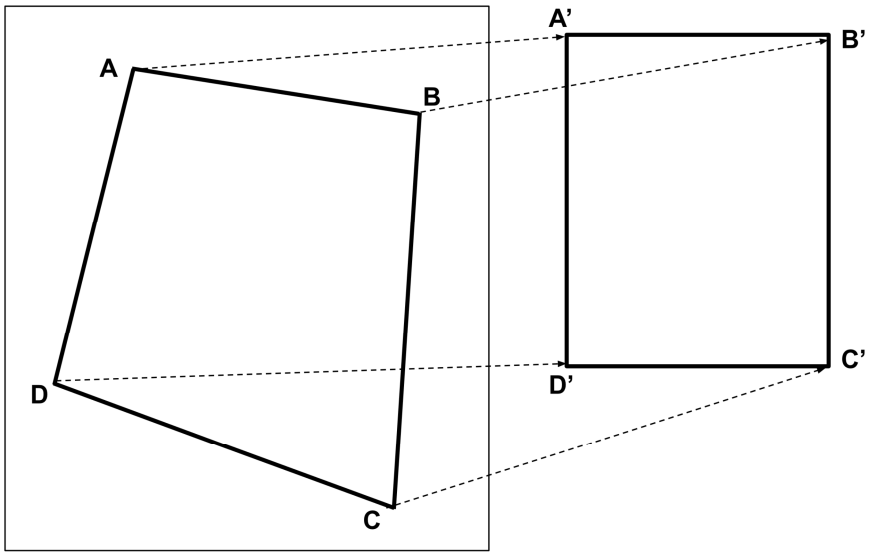


Figure 5.19 Perspective Warping

OpenCV documents have descriptions of the algorithms for image geometric transformations, we do not focus on that in this book, please reference them, if interested, at:

https://docs.opencv.org/4.7.0/da/d54/group__imgproc__transform.html

Perspective warping is commonly used in computer vision and image processing applications, such as image correction, where in many cases images can be distorted due to the angle of the camera, by applying perspective warping the

image can be corrected and aligned properly. Another example is image stitching where multiple images are combined to create a larger panorama, warping can be used to align the images properly so that they fit seamlessly.

In this section, as an example in Figure 5.20 below, the left picture is taken by a tablet camera from the homepage of OpenCV.org, the picture looks distorted, the perspective warping will be used to correct it and make it aligned properly, the result will be shown in the right side of Figure 5.20.

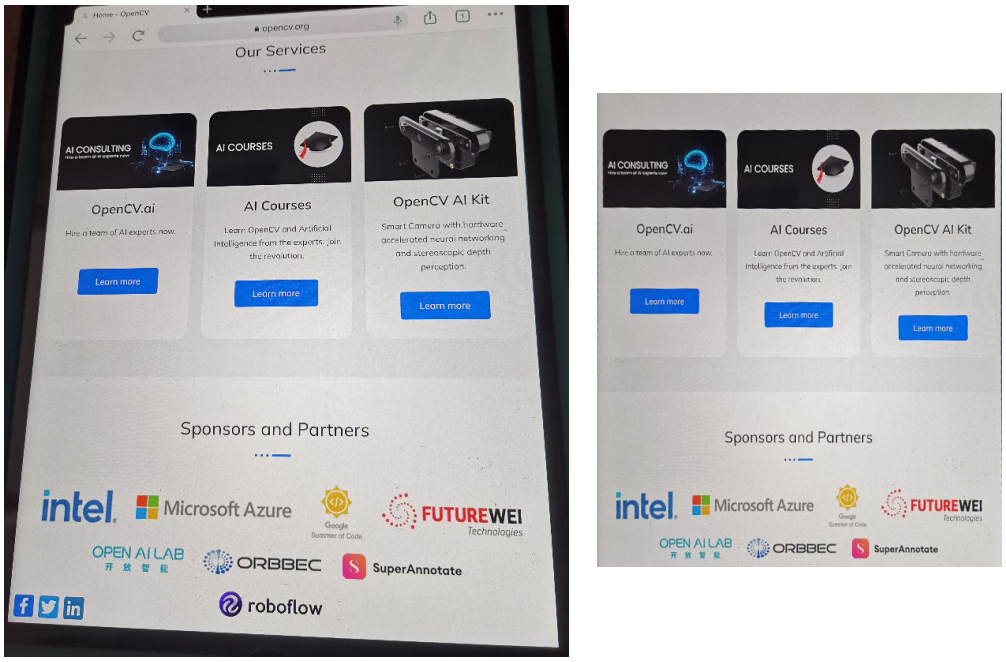


Figure 5.20 An Example of Image Perspective Warping

The picture is taken from the homepage of OpenCV.org

OpenCV provides two functions to perform perspective warping, `cv2.getPerspectiveTransform()` is to calculate the transformation matrix, it takes the four source points and four target points as input. Then `cv2.warpPerspective()` is to perform the perspective warping, it takes the original image, the transformation matrix and the output size.

We will take the four source points from the original image, which indicate the area to transform, and then use them to do the perspective warping.

In the `ImageProcessing` class, add the function `perspective_ warp()`:

```
1  def perspective_warp(self, points, width, height
```

```
                          image=None):
2     if image is None:
3       image = self.image
4     pts_source = np.float32([points[0], points[1],
                               points[3], points[2]])
5     pts_target = np.float32([[0, 0],[width, 0],
                              [0,height],[width,height]])
6     matrix = cv2.getPerspectiveTransform(
                             pts_source, pts_target)
7     result = cv2.warpPerspective(
                             image,matrix,(width,height))
8     return result
```

Explanations:

Line 1	*Define the function for perspective warping, the parameters:* `points`*: coordinates of the four points from the source image* `width/height`*: the width and height of the warping image. Image: the source image, if* `None` *then use the default image of the* `ImageProcessing` *class.*
Line 4	*Define the coordinates of four points for the source.*
Line 5	*Define the coordinates of four points for the target.*
Line 6	*Use* `cv2.getPerspectiveTransform()` *function to generate the transformation matrix, pass the coordinates of source points and target points as parameters.*
Line 7	*Use* `cv2.warpPerspective()` *to perform the perspective transformation.*

Now let's look at the codes for image warping, open *WarpImage.py* file from the Github repository. Remember in section 4.3 we used the mouse to draw polygons, every time when left click happens, a point is added to an array; on right click, the drawing is finished, and the polygon is shown, the points in the array are its vertices. Now we reuse the codes to collect the four source points by slightly modifying them, a point is added to the array on left-click, when the number of

points is counted to 4, it is finished and a red polygon is drawn to indicate the selected area, at the same time call above `perspective_warp()` function to obtain the warped image, and show the result.

Here are the source codes,

```
1   import cv2
2   import numpy as np
3   import common.Draw as dw
4   import common.ImageProcessing as ip
5   drawing = False
6   final_color = (0, 0, 255)
7   drawing_color = (0, 0, 125)
8   width, height = 320, 480
9   points = []
10  def on_mouse(event, x, y, flags, param):
11    global points, drawing, img, img_bk, iproc, warped_image
12    if event == cv2.EVENT_LBUTTONDOWN:
13      drawing = True
14      add_point(points, (x, y))
15      if len(points) == 4:
16        draw_polygon(img, points, (x, y), True)
17        drawing = False
18        img_bk = iproc.copy()
19        warped_image = iproc.perspective_warp(points,
                                                width, height)
20        points.clear()
21        iproc.show("Perspective Warping", image=warped_image)
22    elif event == cv2.EVENT_MOUSEMOVE:
```

```
23      if drawing == True:
24        img = img_bk.copy()
25        draw_polygon(img, points, (x, y))
26
27  def add_point(points, curt_pt):
28    print("Adding point #%d with position(%d,%d)"
29          % (len(points), curt_pt[0], curt_pt[1]))
30    points.append(curt_pt)
31
32  def draw_polygon(img, points, curt_pt, is_final=False):
33    if (len(points) > 0):
34      if is_final == False:
35        dw.draw_polylines(img, np.array([points]),
                            False, final_color)
36        dw.draw_line(img, points[-1], curt_pt,
                            drawing_color)
37      else:
38        dw.draw_polylines(img, np.array([points]),
                            True, final_color)
39      for point in points:
40          dw.draw_circle(img, point, 2, final_color, 2)
41          dw.draw_text(img, str(point), point,
                        color=final_color,
                        font_scale=0.5)
42
43  def print_instruction(img):
44    txtInstruction = "Left click to specify four
                       points to warp image.
                       ESC to exit, 's' to save"
45    dw.draw_text(img,txtInstruction, (10, 20), 0.5,
```

```
                                        (255, 255, 255))
46      print(txtInstruction)
47
48  if __name__ == "__main__":
49      global img, img_bk, iproc, warped_image
50      title = "Original Image"
51      iproc = ip.ImageProcessing(title,
                                   "../res/skewed_image001.jpg")
52      img = iproc.image
53      print_instruction(img)
54      img_bk = iproc.copy()
55      cv2.setMouseCallback(title, on_mouse)
56      iproc.show()
57      while True:
58        iproc.show(image=img)
59          ch = cv2.waitKey(10)
60          if (ch & 0xFF) == 27:
61            break
62          elif ch == ord('s'):
63            # press 's' key to save image
64            filepath = "C:/temp/warp_image.png"
65            cv2.imwrite(filepath, warped_image)
66            print("File saved to " + filepath)
67      cv2.destroyAllWindows()
```

The source codes are not explained line by line here, because it's basically the same as the one for drawing polygons in section 4.3. Execute the codes in *WarpImage.py*, left click on the image to specify the four points in the same way as we did in drawing polygons, the coordinates are shown in the image. After all four

points are collected, it will draw the polygon, and then call `perspective_warp()` function to transform the specified area and show the resulting image.

5.8. Blur Image

```
Source:          BlurImage.py

Library:         common/ImageProcessing.py
```

Blurring an image is a common image processing technique used to reduce noises, smooth out edges, and simplify the image. This section will introduce two types of image blurring techniques, Gaussian Blur and Median Blur.

5.8.1. What is Gaussian Blur

Gaussian Blur is a widely used effect for image processing and is available in many image-editing software. Basically it reduces noises and hides details of the image, and makes the image smoothing.

Figure 5.21 shows what it looks like, the left one is the original image, and the right one is a Gaussian blurred image. By changing the Kernel size, the level of blurring will be changed.

Original Image

Gaussian Blur with kernel=(9,9)

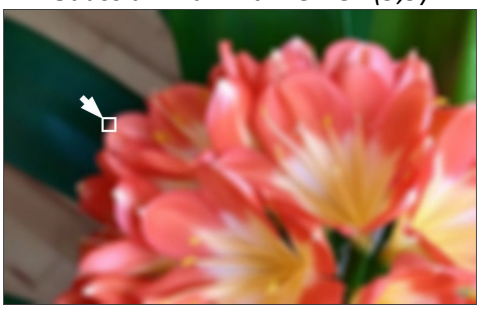


Figure 5.21 Gaussian Blur

Let's take a close look at how the Gaussian blur works, notice the small area highlighted by a white square pointed by a white arrow in Figure 5.21 above. Now zoom in this area and show the details in pixel level, as shown in Figure 5.22, this

square is 9 by 9 pixels. Apply a Gaussian blur on this 9 · 9 area, we get the right-side image in Figure 5.22, the edge of the flower becomes blurred. The size of this area is called *kernel size*, in this example the area is a square of 9 · 9 pixels, then its kernel size is 9. It doesn't have to be a square, it could be a rectangle, say 5 · 9 pixels. However, the kernel size must be odd numbers.

This small area is called *filter*, the Gaussian filter will be applied throughout the entire image starting from the top-left corner, moving from left towards right and from top towards down until the bottom-right corner, in another word the filter swept over the whole image, this is the image blurring process.

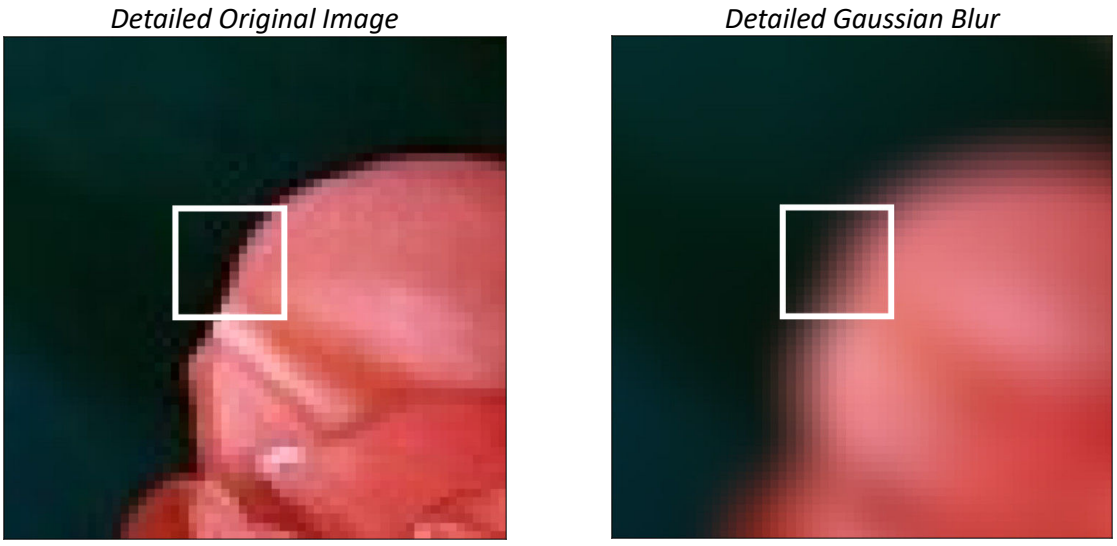


Figure 5.22 Gaussian Blur in Detailed Pixels' Level

Now, let's look at how the Gaussian function works, in other words how the filter is calculated and applied to the area.

Gaussian filter does not simply calculate the average value of the area, in this case the 9 · 9 area. Instead, it calculates a weighted average of the value for this area, the pixels near the center get more weights, and the pixels far away from the center get less weights. And the calculation is done on a channel-by-channel basis, which means it calculates the blue, green and red channels respectively.

Let's dig a little deeper and see how the Gaussian function works, this is the formula in one dimension:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

However, a pixel is determined by x and y coordinates in an image, so the two-dimensional Gaussian formula should be used for image processing, it's shown as Figure 5.23.

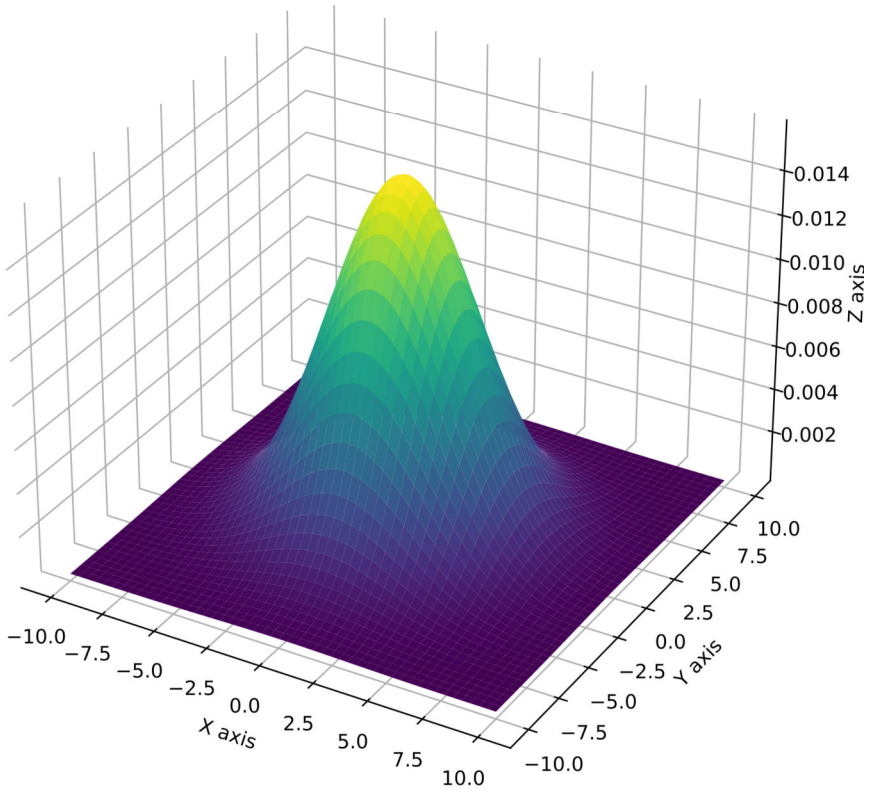


Figure 5.23 Two-Dimensional Gaussian Function

This is generated by the source codes at common/gaussian.py

The two-dimensional Gaussian formula is:

$$p(x,y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Now imagine the Gaussian filter mentioned above, in our case an area of 9 x 9 pixels, is overlaying on the vertex of the above plot, the height is the weight for calculation. The pixels near the center are more important and then get more weights, the pixels near the edge are less important and then get less weights.

This is a basic idea of how Gaussian blur works in general, although the algorithm is not as simple as described above. Fortunately, OpenCV provides a function for this

purpose, `cv2.GaussianBlur()`, and all the complexities are hidden behind the scenes. All we need to do is to call this function and specify the kernel size in width and height, which doesn't have to be the same, but must be odd numbers.

5.8.2. Gaussian Blur

Now we add the Gaussian blur function to the `ImageProcessing` class,

```
1  def blur(self, ksize=(1,1), image=None):
2    if image is None:
3      image = self.image
4    if ksize[0] % 2 == 0:
5      ksize = (ksize[0] + 1, ksize[1])
6    if ksize[1] % 2 == 0:
7      ksize = (ksize[0], ksize[1] + 1)
8    result = cv2.GaussianBlur(image,ksize,
                               cv2.BORDER_DEFAULT)
9    return result
```

Explanations:

Line 1	*Define the `blur()` function, pass `ksize` as parameter.*
Line 4 – 7	*`ksize` must be odd numbers, check `ksize` if not odd then change it to odd.*
Line 8	*Invoke `cv2.GaussianBlur()` function.*

BlurImage.py file has the source codes to perform the Gaussian blur, a trackbar is added to change the kernel size, by changing the kernel size we can observe how the blurring effects are different. In this example a square kernel is used, meaning the width and height are equal. Feel free to modify the codes to use a rectangle kernel and observe the blurring effects.

```
1  import cv2
2  import common.ImageProcessing as ip
```

```
3
4   def change_ksize(value):
5       global ksize
6       if value % 2 == 0:
7           ksize = (value+1, value+1)
8       else:
9           ksize = (value, value)
10
11  if __name__ == "__main__":
12      global ksize
13      ksize = (5,5)
14      iproc = ip.ImageProcessing("Original",
                                   "../res/flower003.jpg")
15      iproc.show()
16
17      cv2.namedWindow("Gaussian Blur")
18      cv2.createTrackbar("K-Size", "Blur", 5, 21,
                           change_ksize)
19
20      while True:
21          blur = iproc.blur(ksize)
22          iproc.show("Blur", blur)
23
24          ch = cv2.waitKey(10)
25          # Press 'ESC' to exit
            if (ch & 0xFF) == 27:
26              break
27          # Press 's' to save
```

```
            elif ch == ord('s'):
28              filepath = "C:/temp/blend_image.png"
29              cv2.imwrite(filepath, blur)
30              print("File saved to " + filepath)
31      cv2.destroyAllWindows()
```

Explanations:

Line 4 – 9	*Trackbar callback function, get kernel size from trackbar, and change it to odd number if it's not.*
Line 14 – 15	*Instantiate the `ImageProcessing` class with an image and show it as original image.*
Line 17 – 18	*Create a trackbar in another window called "Gaussian Blur".*
Line 21	*Call `blur()` function defined above in `ImageProcessing` class, pass the kernel size as a parameter.*
Line 22	*Show the blurred image in the "Gaussian Blur" window.*

Figure 5.24 is the result, as the `K-Size` trackbar is changing, the degree of blurring is also changing in real-time.

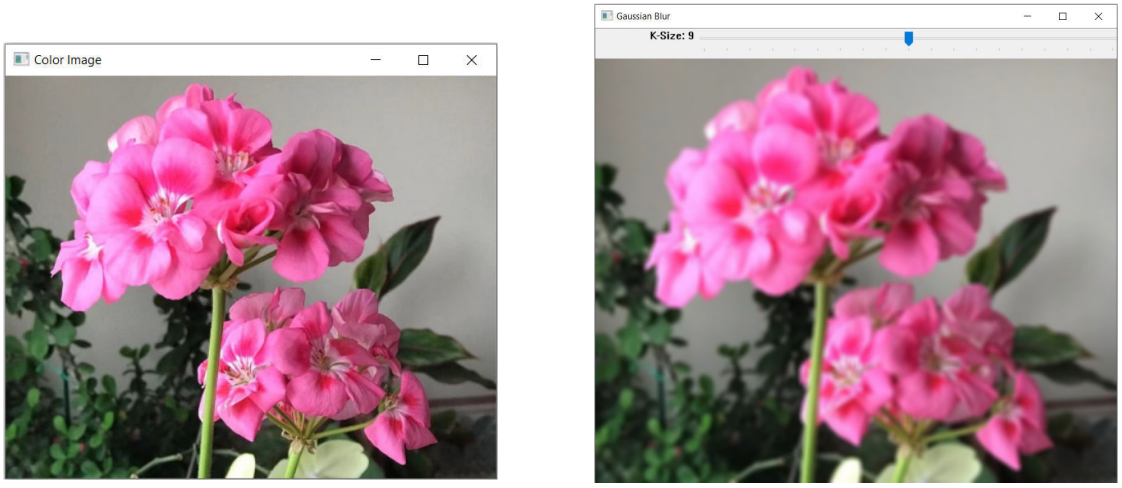


Figure 5.24 Gaussian Blur Results

5.8.3. Median Blur

Same as Gaussian Blur, the Median Blur is also widely used in image processing, it is often used for noise reduction purposes.

Similar to the Gaussian Blur filter, instead of applying Gaussian formula, the Median Blur calculates the median of all the pixels inside the kernel filter and the central pixel is replaced with this median value. OpenCV also provides a function for this purpose, `cv2.medianBlur()`.

Here is the code in `ImageProcessing` class to apply the median blur function,

```
1  def median_blur(self, ksize=1, image=None):
2    if image is None:
3      image = self.image
4    result = cv2.medianBlur(image, ksize)
5    return result
```

In *BlurImage.py* file the codes for Median Blur are quite similar to the Gaussian Blur, a trackbar can change the kernel size, and the effects can be observed in real-time. Below shows the original image vs. the median-blurred image.

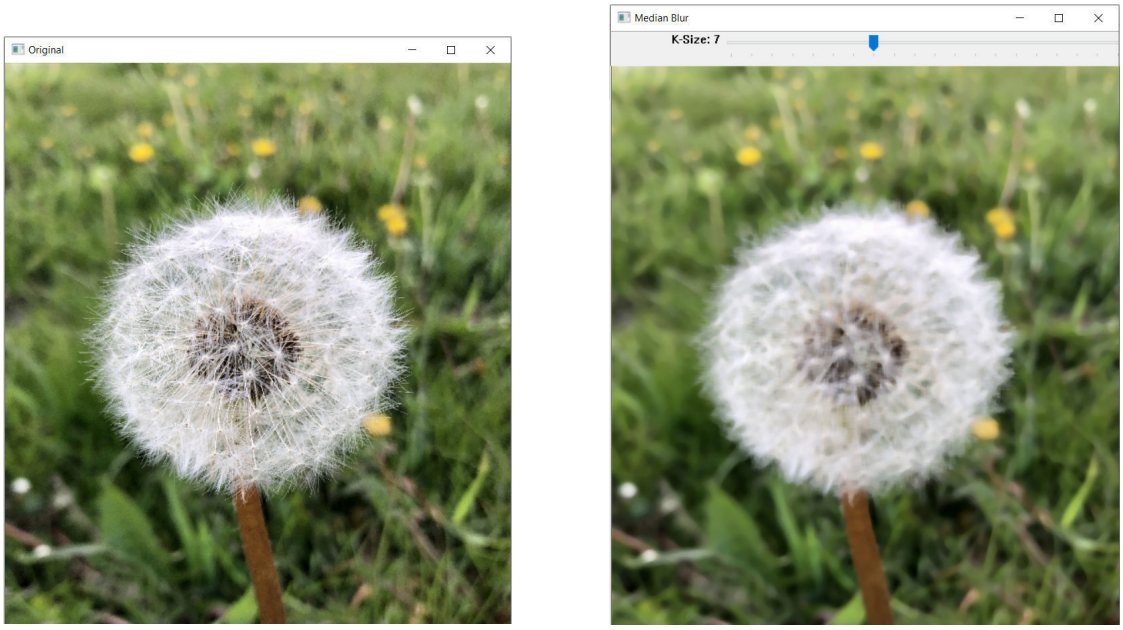


Figure 5.25 Median Blur Results

5.9. Histogram

```
Source:          Histogram.py

Library:         common/ImageProcessing.py
```

5.9.1. About Histogram

A histogram is a graphical representation of the distribution of pixel values in an image. In image processing, a histogram can be used to analyze the brightness, contrast, and overall intensity of an image. It is plotted in a x-y chart, the x-axis of a histogram represents the pixel values ranging from 0 to 255, while the y-axis represents the number of pixels in the image that have a given value. A histogram can show whether an image is predominantly dark or light, and whether it has high or low contrast. It can also be used to identify any outliers or unusual pixel values.

To better understand the histogram, plot an image and draw some squares and rectangles and fill them with the color value of 0, 50, 100, 150, 175, 200 and 255, as shown in the left-side of Figure 5.26. The coordinates of each square/rectangle are also displayed in the image, so we can easily calculate how many pixels for each color.

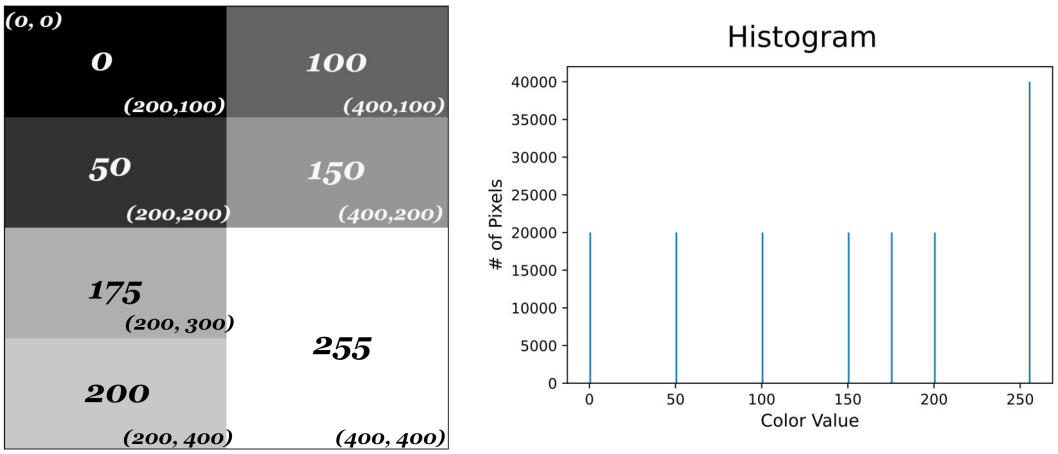


Figure 5.26 Histogram

In the histogram plot in the right-side, x-axis is the color value from 0 to 255, and y-axis is the number of pixels. The y value is 20,000 at x=0, meaning there are

20,000 pixels that have a color value of 0, from the left-side image we can see the black rectangle (color: 0) is from (0, 0) to (200, 100), the total number of black pixels is 200 x 100 = 20,000. The histogram plot also shows the color value 0 has 20,000 pixels. The white square (color: 255) has 40,000 pixels. In the same way, calculate the number of pixels for other colors, there are 20,000 pixels for color values of 50, 100, 150, 175 and 200.

This is how the histogram works, the number of pixels and the color values are shown in the histogram.

Below is the code to create the above image and histogram plot.

```
1   def show_histogram():
2     img = np.zeros((400, 400), np.uint8)
3     cv2.rectangle(img, (200,0), (400, 100), (100), -1)
4     cv2.rectangle(img, (0, 100), (200, 200), (50), -1)
5     cv2.rectangle(img, (200, 100), (400, 200), (150), -1)
6     cv2.rectangle(img, (0,200), (200, 300), (175), -1)
7     cv2.rectangle(img, (0, 300), (200, 400), (200), -1)
8     cv2.rectangle(img, (200,200), (400, 400), (255), -1)
9     fig = plt.figure(figsize=(6, 4))
10    fig.suptitle('Histogram', fontsize=20)
11    plt.xlabel('Color Value', fontsize=12)
12    plt.ylabel('# of Pixels', fontsize=12)
13    plt.hist(img.ravel(), 256, [0, 256])
14    plt.show()
```

Explanations:

Line 2	*Create a numpy array as a blank canvas with all zeros.*
Line 3 - 8	*Draw squares and rectangles and fill them with specific color values.*
Line 9 – 12	*Define a plot using matplotlib library, set title and X, Y-axis labels.*
Line 13	*Create a histogram using matplotlib function.*
Line 14	*Show the histogram plot*

Histograms can be used for various purposes in image processing, below is a list of some of the use cases,

Image equalization, by modifying the distribution of pixel values in the histogram, it is possible to improve the contrast and overall appearance of an image.

Thresholding, by analyzing the histogram, it is possible to determine the optimal threshold value for separating the foreground and background of an image.

Color balance, by analyzing the histograms of individual color channels, it is possible to adjust the color balance of an image.

In conclusion, histograms are an important tool in image processing for analyzing and manipulating the distribution of pixel values in an image.

5.9.2. Histogram for Grayscale Images

There are two ways to compute and display histograms. First, OpenCV provides `cv2.calcHist()` function to compute a histogram for an image, second, use `matplotlib` to plot the histogram diagram, `matplotlib` is a Python library for creating static, animated, and interactive visualizations.

Figure 5.27 shows the histogram of a real image.

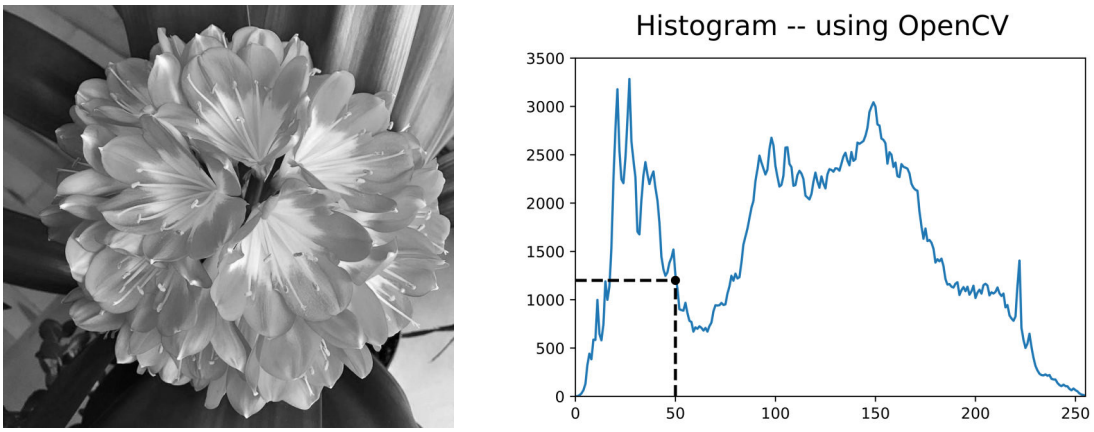


Figure 5.27 Histogram for a Grayscale Image

Look at a specific point, i.e the • point in the histogram plot at the right-side of Figure 5.27, it means there are about 1,200 pixels with color value of 50 in the left-side grayscale image.

This is how to read the histogram diagram which gives an overall idea of how the color value is distributed.

Here are the codes to produce the histogram in Figure 5.27:

```
1   # Get histogram using OpenCV
```

```
2   def show_histogram_gray(image):
3       hist = cv2.calcHist([image], [0], None, [256], [0, 256])
4       fig = plt.figure(figsize=(6, 4))
5       fig.suptitle('Histogram - using OpenCV', fontsize=18)
6       plt.plot(hist)
7       plt.show()
```

The second way to display a histogram is to use `matplotlib`, which provides `plt.hist()` function to generate a histogram plot, it does the exact same thing just looks a little bit differently, as Figure 5.28:

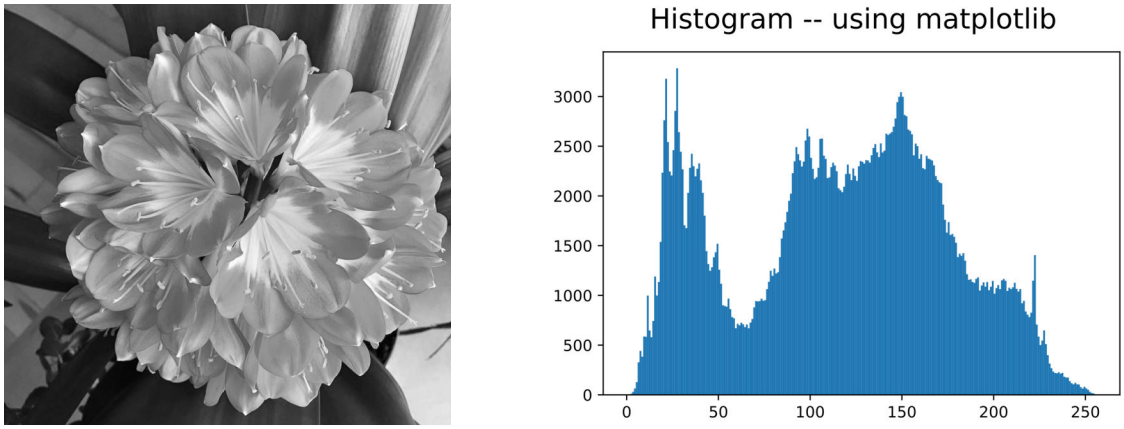


Figure 5.28 Histogram for a Grayscale Image

Here are the codes to produce the histogram in Figure 5.28,

```
9   # Alternative way for histogram using matplotlib
10  def show_histogram_gray_alt(image):
11      fig = plt.figure(figsize=(6, 4))
12      fig.suptitle('Histogram - using matplotlib',fontsize=18)
13      plt.hist(image.ravel(), 256, [0, 256])
14      plt.show()
```

Explanations:

Line 1 - 7	*Use OpenCV cv2.calchist to generate a histogram.*
Line 3	*Call `cv2.calcHist()` function, pass the image as the parameter.*
Line 4 - 6	*Create a plot using `matplotlib`, specify the plot size, set the title, and plot the histogram created in line 3.*
Line 7	*Show the plot*
Line 10 - 14	*Alternatively, use `matplotlib` function to generate a histogram.*
Line 11 -12	*Create a plot using `matplotlib`, specify the plot size, set the title.*
Line 13	*Call `plt.hist()` function to create a histogram of the image.*

5.9.3. Histogram for Color Images

The histogram is created on a channel-by-channel basis, a color image has blue, green and red channels. To plot the histogram for color images, the image should be split into blue, green and red channels, and plot the histogram one by one. With `matplotlib` library we can put the three histograms in one plot.

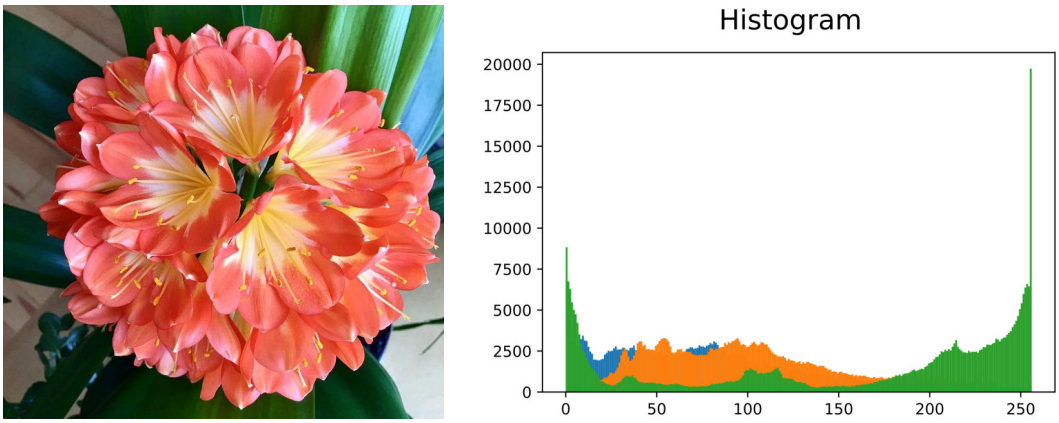


Figure 5.29 Histogram for a Color Image

Below is the code to generate above histogram for color image:

```
1   def show_histogram_color(image):
2       blue, green, red = cv2.split(image)
```

```
3       # cv2.imshow("blue", blue)
4       # cv2.imshow("green", green)
5       # cv2.imshow("red", red)
6       fig = plt.figure(figsize=(6, 4))
7       fig.suptitle('Histogram', fontsize=18)
8       plt.hist(blue.ravel(), 256, [0, 256])
9       plt.hist(green.ravel(), 256, [0, 256])
10      plt.hist(red.ravel(), 256, [0, 256])
11      plt.show()
```

Explanations:

Line 2	*Split the color image into blue, green and red channels*
Line 3 - 5	*Optionally show the blue, green and red channels.*
Line 6 - 7	*Create a plot, set the title.*
Line 8 - 10	*Add the histograms for blue, green and red to the plot*
Line 11	*Show the histogram plot.*

In summary, the shape of the histogram can provide information about the overall color distributions, as well as valuable insights into the overall brightness and contrast of an image. If the histogram is skewed towards the higher intensity levels, the image will be brighter, while a skew towards lower intensity levels indicates a darker image. A bell-shaped histogram indicates a well-balanced contrast.

Histogram equalization is a common technique used to enhance the contrast of an image by redistributing the pixel intensities across a wider range. This technique can be used to improve the visual quality of images for various applications, such as medical imaging, satellite imaging, and digital photography.