CHAPTER 9

Image and Video Processing

In Chapter 8, you learned about audio processing in GNU Octave in detail. In this chapter, you will use some of the concepts you learned about in Chapter 7 to understand image processing.

In this chapter, you will learn about the following list of topics:

- Image processing
- Video processing

Image Processing

With the growing availability of good cameras in phones over the past few years, and the outbreak of social media platforms like Instagram, YouTube, etc. users all over the world can now upload visually pleasing images and videos. All photo editing applications, like **Photoshop** or **GIMP**, used for this purpose employ image processing. In this section, you will learn how to process digital images with GNU Octave.

You will first look at the basic installation required to work with images and then move on to writing your own code to work with them.

Similar to previous chapters, you must install the image package available at `https://octave.sourceforge.io/image/index.html`.

© Ashwin Pajankar and Sharvani Chandu 2020
A. Pajankar and S. Chandu, *GNU Octave by Example*,
https://doi.org/10.1007/978-1-4842-6086-9_9

You will use a Jupyter notebook for all of the demonstrations in this chapter. Create a new notebook for this chapter. In a new cell in the Jupyter notebook, run the following command:

```
pkg install -forge image
```

Next, load the image package by running the following command:

```
pkg load image
```

Let's first explore how to read and write images. For this, download any image from the web or use any image on your computer, and save it in the current folder of your Jupyter notebook with the name **sample_color.jpg**. We will use the image in Figure 9-1 to demonstrate the results in this chapter.

Figure 9-1. *A sample image*

Loading, Displaying, and Resizing Images

Now you'll learn how to load the images into Octave. Type the following command into a new cell in the notebook:

```
color_image = imread('sample_color.jpg');
```

`imread` loads the image and stores it to a variable, in this case `color_image`. Now display the image:

```
%plot gnuplot
figure(1), imshow(color_image);
```

This will display the image in a new window. If you zoom into the image, you will notice the image looks like small squares, as you can see in Figure 9-2.

Figure 9-2. *Zooming into the flower image*

The reason you see the small squares in the image is because the image is stored as three-dimensional matrix and each member is an 8-bit unsigned integer (uint8).

Let's get the size of the image:

```
size(color_image)
```

You will see something like this in the output:

```
ans = 3648 5472 3
```

In this case, the image is a matrix of dimensions 3648 X 5472 X 3.

You can resize the image using `imresize`:

```
resized_image = imresize(color_image, [512, 512]);
```

Here you resize the image to 512 X 512 X 3. Check this for yourself using `size`. When using `imresize()`, you can either give a scale to which you want to resize the dimensions or directly mention the size to which you want to resize, as you have done here.

Color Space

Now let's explore how the data is stored and how the image obtains its color. You have seen that the size of an image has three dimensions and the third dimension has the value 3. This is true for all color images. Each 2D matrix of the third dimension is called a channel. The first is for red (R), the second is for green (G), and the third is for blue (B). You must be familiar with the acronym RGB; this comes from the channel names.

A color space is a specific way of organizing colors such that they can be reproducible in digital representation. With a triplet of each value corresponding to the intensity in R, G, and B colors, you can cover most of the colors that the human eye can perceive.

Now let's explore the concept visually. In a new cell, type the following code:

```
red_image = color_image;
red_image(:,:,2) = 0;
red_image(:,:,3) = 0;

%plot gnuplot
figure(2), imshow(red_image);
```

In this code, you copy the `color_image` to `red_image` and then set the green and blue channel values to 0. The image is shown in Figure 9-3. Only the red component of the image is visible in the image. You can try for yourself for the other two channels or with a combination of two channels.

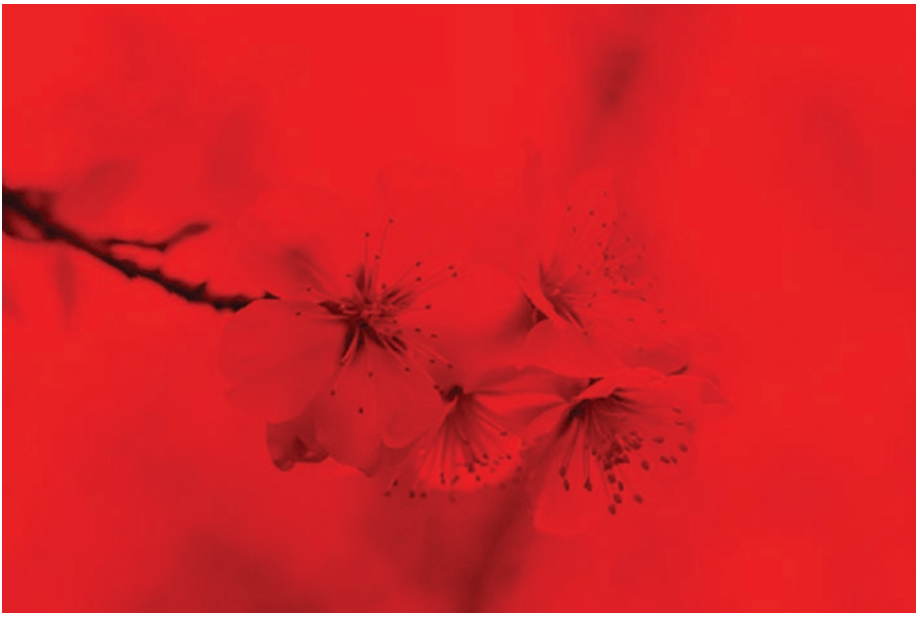

Figure 9-3. *Red channel of the flower image*

You are all familiar with old pictures or movies that are monochromatic. So, if having only one channel is displaying the image in that color space, then how do you get monochromatic images? Before we jump into that, let's look at the data stored in the image. For that, let's display the values in a small portion of the image like this:

```
color_image(1:10, 1:10, 2)
```

You will see something like this in the Jupyter notebook (not the exact same values, because they will depend on your image):

```
ans =130 125 125 125 124 124 126 126 124 123 121
127 126 125 124 121 123 125 124 123 126 124
126 126 125 126 123 123 124 124 124 124 123
126 127 127 128 126 125 123 124 125 123 122
126 127 126 127 126 126 123 123 123 122 121
125 123 123 124 126 126 124 122 121 122 121
125 124 123 123 125 125 123 121 121 123 121
125 125 125 125 124 123 122 122 121 125 122
125 127 126 126 123 122 121 124 122 124 122
123 129 127 125 124 122 123 124 122 120 120
124 124 124 125 125 123 124 124 124 124 123
```

Note that the values are between 0-255. This is because the image is stored with an uint8 datatype and it has range of 0-255 (2^8=256), as discussed.

To get a monochromatic image, the three channels are combined to one. You usually do this using the following function:

```
gray_image = rgb2gray(color_image);

%plot gnuplot
figure(3), imshow(gray_image);
```

The output is shown in Figure 9-4.

Figure 9-4. *Gray scale image of the flower*

You can see the display of the monochromatic image. This is also called a grayscale image.

Now let's see the size of the `gray_image`:

```
size(gray_image)
ans = 3648 5472
```

You will notice that the size of the `gray_image` is the same as the size of the `color_image`, except for the three color channels.

There are other color spaces, which you can explore by yourself. Some of the important ones are RGB and HSV (Hue-Saturation-Value).

Cropping, Rotating, and Saving Images

You are all familiar with basic photo viewing or editing tools that lets us crop or flip images. In this section, you will explore these cool features and then learn how to save an image.

Let's first look at cropping. In a new cell, run the following code:

```
crop_image = color_image(2000:3000, 2000:4000, :);

%plot gnuplot
figure(4), imshow(crop_image);
```

Make sure that you do not exceed the limits of the image size you are using.

The output is shown in Figure 9-5.

Figure 9-5. *The cropped image*

In this way, you can crop out the portion you want in your own images if you know the desired pixel location.

Let's now look at flipping and rotating images:

```
up_down_flip_image = flipud(color_image);

%plot gnuplot
figure(5), imshow(up_down_flip_image);
```

This code flips the image along the horizontal, as shown in Figure 9-6.

Figure 9-6. *Horizontally flipped image*

Similarly, you can also flip the image along the vertical axis, as shown in Figure 9-7:

```
left_right_flip_image = fliplr(color_image);

%plot gnuplot
figure(6), imshow(left_right_flip_image);
```

Figure 9-7. *Vertically flipped image*

You can also do a flip both horizontally and vertically like this:

```
flip_image = fliplr(flipud(color_image));

%plot gnuplot
figure(7), imshow(flip_image);
```

You can do the same using `imrotate()`, like this:

```
rotated_image = imrotate(color_image, 180);

%plot gnuplot
figure(8), imshow(rotated_image);
```

Here, you rotated the image by 180 degrees to get the same image as shown in Figure 9-8 as the previous code. You can try for yourself with different angles in `imrotate()`.

Figure 9-8. *Vertically and horizontally flipped image or image rotated by 180 degrees*

If you want to save any of the images you modified, you can do so using `imwrite`:

```
imwrite(rotated_image, 'flipped_image.jpg');
```

The first parameter to the function is the image you want to save and the second one is the string with the path to the image you want to save along with the image name.

FFT2

In Chapter 7, we discussed FFT (Fast Fourier Transform). In this section, you will look at the Fourier transform for images. In images, frequency corresponds to how fast the pixel intensity changes. When there are fast

changes, it is a high frequency region; if little changes, it is a low frequency region. The applications of the concepts you study here form the basics of low-pass filtering/smoothing and high-pass filtering/edge detection, which are the fundamentals of many advanced image processing techniques. You can explore more on your own once you are clear on the fundamentals.

FFT2 computes a discrete Fourier transform on the 2D matrix. For this, first create a 2D pulse image:

```
pulse_2d = zeros(500, 500, 3);
pulse_2d(246:255, 246:255, :) = 255;
pulse_2d = im2bw(pulse_2d);
```

This will generate a 2D pulse image as shown in Figure 9-9.

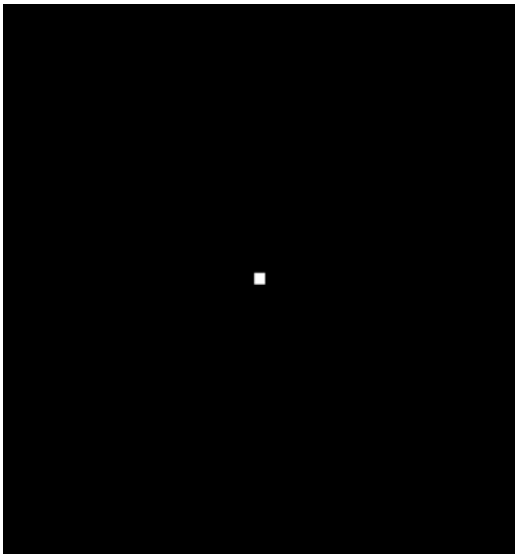

Figure 9-9. *2D pulse*

Let's now compute the Fourier transform on this image:

```
pulse_freq = fft2(pulse_2d);
```

Before displaying the image, you need to first get the absolute value of the frequency and then do any fftshit to align the center to the center of the image. Recall that this is similar to the function `sinc(), which` you saw in a previous chapter, extended to 2D, where the peaks of the sinc function are white with a maximum value and the valleys of the sinc function are black with a minimum value, as shown in Figure 9-10.

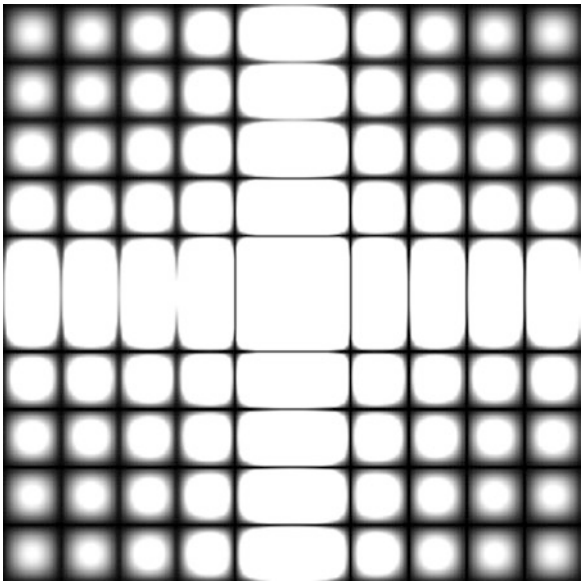

Figure 9-10. *Fourier transform of 2D pulse*

Video Processing

In this section, you will explore the basic workings of video processing. Normally, videos are conceptually visualized as a 4D object, the fourth dimension being time. Imagine it to be something like Figure 9-11.

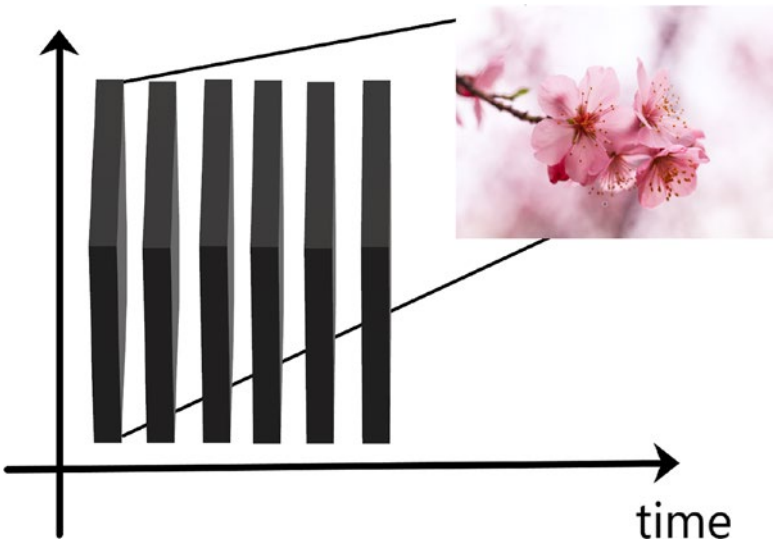

Figure 9-11. *Visual interpretation of videos*

First, to work with videos, you need to install the video package. Follow the steps similar to what you did to install the image package:

```
pkg install -forge video
pkg load video
```

Now generate a video and write it to a video file:

```
w = VideoWriter("images.mp4");
open(w);
for i = 1:360
    img = imrotate(color_image, i);
    img = imresize(img, [512, 512]);
    writeVideo(w, img);
endfor
close(w);
```

In this code, you first create a video writer and then you open the file.

For your learning purposes, you are utilizing functions you learned in the previous sections of this chapter.

You use a `for` loop to rotate the image and resize it. And then you write each frame into the video file. Resizing images is important because all of the image frames in a video should be of the same dimensions, as shown in Figure 9-11. You can use any video format that is supported by Octave; this demonstration uses the **.mp4** format.

Here's how to read the video file:

```
w = VideoReader ("images.mp4");
while (!isempty(img = readFrame(x)))
    imshow(img);
endwhile
```

This code reads each frame from the video file and then displays it using `imshow`. You can explore more advanced techniques using what you learned in this section.

Summary

In this chapter, you learned how to read, save, and display images. You also learned about color spaces of images, plus cropping, flipping, and rotating images. You looked at a Fourier transform on images. You also learned how to read and write videos.

The next part is the Appendix. It covers several small topics that could not find place in the previous chapters.