CHAPTER 8

GRAY-LEVEL TRANSFORMATIONS

WHAT WILL WE LEARN?

- What does it mean to "enhance" an image?
- How can image enhancement be achieved using gray-level transformations?
- What are the most commonly used gray-level transformations and how can they be implemented using MATLAB?

8.1 INTRODUCTION

This chapter—and also Chapters 9 and 10—will discuss the topic of *image enhancement* in the spatial domain. As discussed in Chapter 1, image enhancement techniques usually have one of these two goals:

1. To improve the subjective quality of an image for human viewing.
2. To modify the image in such a way as to make it more suitable for further analysis and automatic extraction of its contents.

In the first case, the ultimate goal is an improved version of the original image, whose interpretation will be left to a human expert—for example, an enhanced X-ray image that will be used by a medical doctor to evaluate the possibility of a fractured bone. In the second scenario, the goal is to serve as an intermediate step toward an automated solution that will be able to derive the semantic contents of the image—for

Practical Image and Video Processing Using MATLAB®. By Oge Marques.

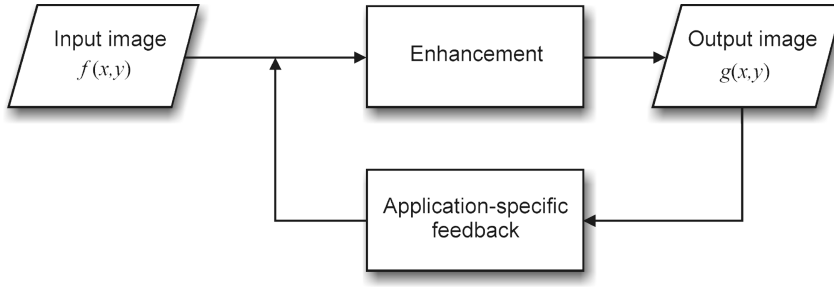

FIGURE 8.1 The image enhancement process. Adapted and redrawn from [Umb05].

example, by improving the contrast between characters and background on a page of text before it is examined by an OCR algorithm. Sometimes these goals can be at odds with each other. For example, sharpening an image to allow inspection of additional fine-grained details is usually desired for human viewing, whereas blurring an image to reduce the amount of irrelevant information is often preferred in the preprocessing steps of a machine vision solution.

Another way to put it is to say that image enhancement techniques are used when either (1) an image needs improvement, or (2) the low-level features must be detected [SS01]. Image enhancement methods "improve the detectability of important image details or objects by man or machine" [SS01]. This is different than attempting to restore a degraded image to its original (or *ideal*) condition, which is the scope of image restoration techniques (Chapter 12).

It is important to mention that image enhancement algorithms are usually *goal specific*, which implies that there is no general theory of image enhancement nor is there a universal enhancement algorithm that always performs satisfactorily. Rather, it is typically an interactive process in which different techniques and algorithms are tried and parameters are fine-tuned, until an acceptable result is obtained (Figure 8.1). Moreover, it is often a subjective process in which the human observer's specialized skills and prior problem-domain knowledge play a significant role (e.g., a radiology expert may find significant differences in the quality of two X-ray images that would go undetected by a lay person).

This chapter focuses on *point operations* whose common goal is to enhance an input image. In some cases, the enhancement results are clearly targeted at a human viewer (e.g., contrast adjustment techniques, Section 8.3.1), while in some other cases the results may be more suitable for subsequent stages of processing in a machine vision system (e.g., image negative, Section 8.3.2).

8.2 OVERVIEW OF GRAY-LEVEL (POINT) TRANSFORMATIONS

Point operations (briefly introduced in Section 2.4.1) are also referred to as *gray-level transformations* or *spatial transformations*. They can be expressed as

$$g(x, y) = T\left[f(x, y)\right] \tag{8.1}$$

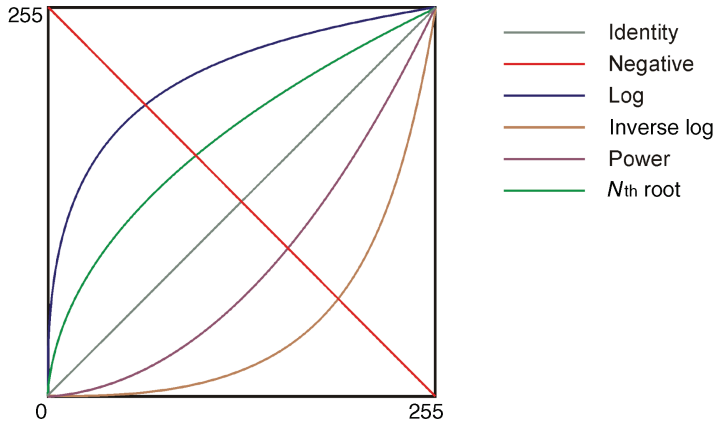

FIGURE 8.2 Basic gray-level transformation functions.

where $g(x, y)$ is the processed image, $f(x, y)$ is the original image, and T is an operator on $f(x, y)$.

Since the actual coordinates do not play any role in the way the transformation function processes the original image,[1] equation (8.1) can be rewritten as

$$s = T[r] \tag{8.2}$$

where r is the original gray level of a pixel and s is the resulting gray level after processing.

Point transformations may be *linear* (e.g., negative), *piecewise linear* (e.g., gray-level slicing), or *nonlinear* (e.g., gamma correction). Figure 8.2 shows examples of basic linear (*identity* and *negative*) and nonlinear (*log*, *inverse log*, *power*—also known as *gamma*—, and *n*th *root*) transformation functions.

Point operations are usually treated as simple mapping operations whereby the new pixel value at a certain location (x_0, y_0) depends only on the original pixel value at the same location and the mapping function. In other words, the resulting image does not exhibit any change in size, geometry, or local structure if compared with the original image.[2]

In this book, we will call *linear* point transformations those that can be mathematically described by a single linear equation:

$$s = c \cdot r + b \tag{8.3}$$

where r is the original pixel value, s is the resulting pixel value, and c is a constant—responsible for controlling the contrast of the output image—, whereas b is another constant whose value impacts the output image's overall brightness.

[1] If the transformation function $T(.)$ is independent of the image coordinates, the operation is called *homogeneous*.

[2] This is in clear contrast with the operations described in Chapter 7.

From a graphical perspective, a plot of s as a function of r will show a straight line, whose slope (or gradient) is determined by the constant c; the constant term b determines the point at which the line crosses the y-axis. Since the output image should not contain any pixel values outside a certain range (e.g., a range of [0, 255] for monochrome images of class `uint8` in MATLAB), the plot usually also shows a second, horizontal, straight line that is indicative of *clamping* the results to keep them within range (see left and middle columns in Figure 8.4 for examples).

If a point transformation function requires several linear equations, one for each interval of gray-level values, we shall call it a *piecewise linear* function (Section 8.3.5).

Transformation functions that cannot be expressed by one or more linear equations will be called *nonlinear*. The power law (Section 8.3.3) and log (Section 8.3.4) transformations are examples of nonlinear functions.

■ EXAMPLE 8.1

Figure 8.4 shows the results of applying three different linear point transformations to the input image in Figure 8.3. Table 8.1 summarizes the values of c and b for each of the three cases as well as the visual effect on the processed image.

FIGURE 8.3 Linear point transformations example: input image.

TABLE 8.1 Examples of Linear Point Transformations (Images and Curves in Figure 8.4)

Column	c	b	Effect on Image
Left	2	32	Overall brightening, including many saturated pixels
Middle	1	−56	Overall darkening
Right	0.3	0	Significant contrast reduction and overall darkening

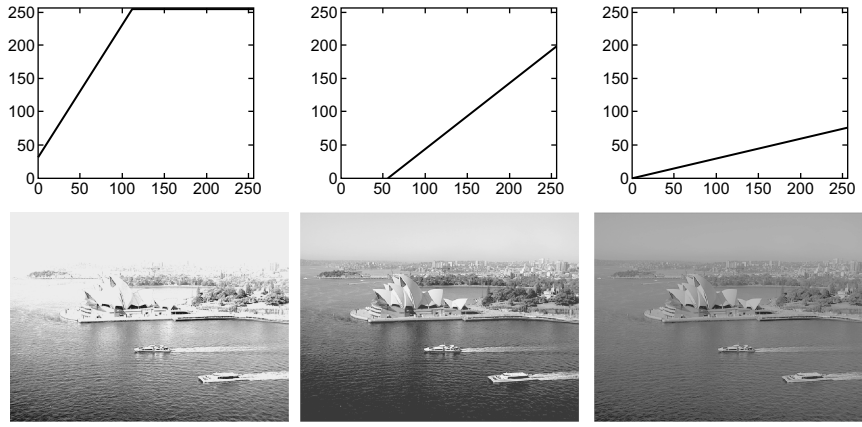

FIGURE 8.4 Linear point transformations and their impact on the overall brightness and contrast of an image: brightening (left), darkening (middle), and contrast reduction (right).

8.3 EXAMPLES OF POINT TRANSFORMATIONS

In this section, we show examples of some of the most widely used point transformations.

8.3.1 Contrast Manipulation

One of the most common applications of point transformation functions is *contrast manipulation* (also known by many other names such as *contrast stretching*, *gray-level stretching*, *contrast adjustment*, and *amplitude scaling*). These functions often exhibit a curve that resembles the curve of a sigmoid function (Figure 8.5a): pixel values of

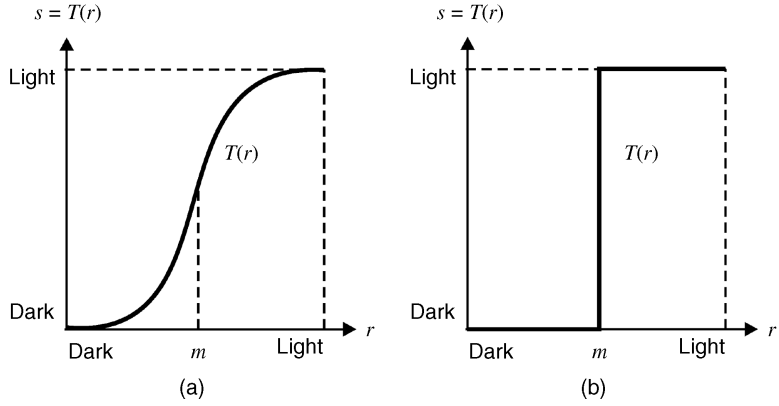

FIGURE 8.5 Examples of gray-level transformations for contrast enhancement. Redrawn from [GW08].

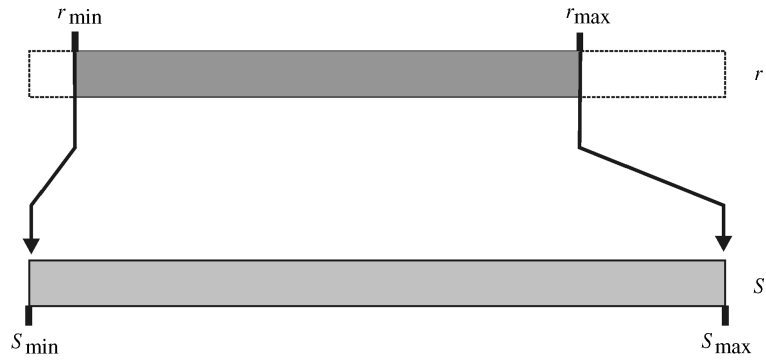

FIGURE 8.6 Autocontrast operation. Redrawn from [BB08].

$f < m$ are compressed toward darker values in the output image, whereas values of $f > m$ are mapped to brighter pixel values in the resulting image. The slope of the curve indicates how dramatic the contrast changes will be; in its most extreme case, a contrast manipulation function degenerates into a binary thresholding[3] function (Figure 8.5b), where pixels in the input image whose value is $f < m$ become black and pixels whose value is $f > m$ are converted to white.

One of the most useful variants of contrast adjustment functions is the *automatic contrast adjustment* (or simply *autocontrast*), a point transformation that—for images of class `uint8` in MATLAB—maps the darkest pixel value in the input image to 0 and the brightest pixel value to 255 and redistributes the intermediate values linearly (Figure 8.6).

The autocontrast function can be described as follows:

$$s = \frac{L-1}{r_{\max} - r_{\min}} \cdot (r - r_{\min}) \tag{8.4}$$

where r is the pixel value in the original image (in the [0, 255] range), $r_{\max}$ and $r_{\min}$ are the values of its brightest and darkest pixels, respectively, s is the resulting pixel value, and $L - 1$ is the highest gray value in the input image (usually $L = 256$). Figure 8.7 shows an example of an image before and after autocontrast.

In MATLAB

MATLAB's IPT has a built-in function `imadjust` to perform contrast adjustments (including autocontrast). You will learn more about it in Tutorial 9.3 (page 195).

In MATLAB, *interactive* brightness and contrast adjustments can also be performed using `imcontrast` that opens the *Adjust Contrast* tool introduced in Chapter 4.

We shall revisit the topic of contrast adjustments in Chapter 9 when we present techniques such as *histogram equalization* and *histogram stretching*.

[3] We shall study thresholding techniques in Chapter 15.

(a) (b)

FIGURE 8.7 (a) Example of an image whose original gray-level range was [90, 162]; (b) the result of applying the autocontrast transformation (equation (8.4)).

8.3.2 Negative

The negative point transformation function (also known as *contrast reverse* [Pra07]) was described in Section 6.1.2. The negative transformation is used to make the output more suitable for the task at hand (e.g., by making it easier to notice interesting details in the image).

In MATLAB

MATLAB's IPT has a built-in function to compute the negative of an image: `imcomplement`, which was used in Tutorial 6.1 (page 113).

8.3.3 Power Law (Gamma) Transformations

The power law transformation function is described by

$$s = c \cdot r^{\gamma} \tag{8.5}$$

where r is the original pixel value, s is the resulting pixel value, c is a scaling constant, and γ is a positive value. Figure 8.8 shows a plot of equation (8.5) for several values of γ.

■ EXAMPLE 8.2

Figure 8.9 shows the results of applying gamma correction to an input image using two different values of γ. It should be clear from the figure that when $\gamma < 1$, the resulting image is darker than the original one; whereas for $\gamma > 1$, the output image is brighter than the input image.

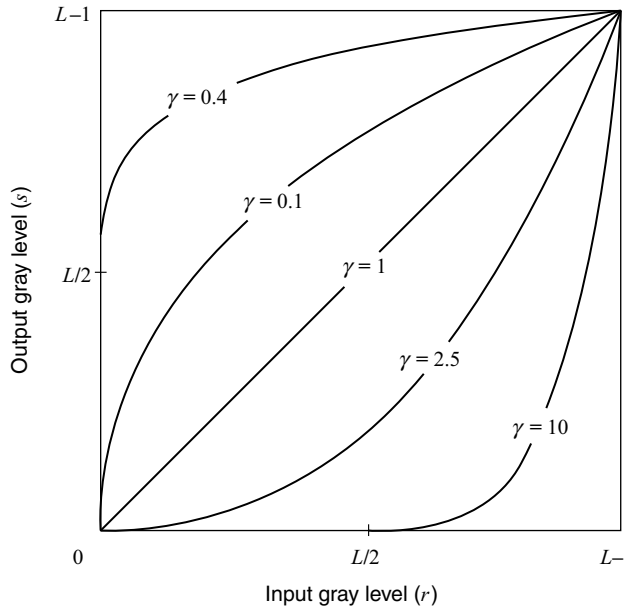

FIGURE 8.8 Examples of power law transformations for different values of γ.

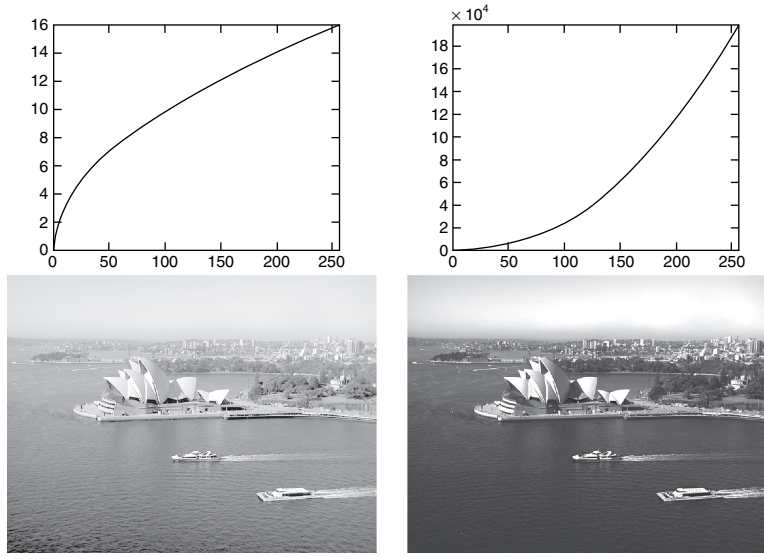

FIGURE 8.9 Examples of gamma correction for two different values of γ: 0.5 (left) and 2.2 (right).

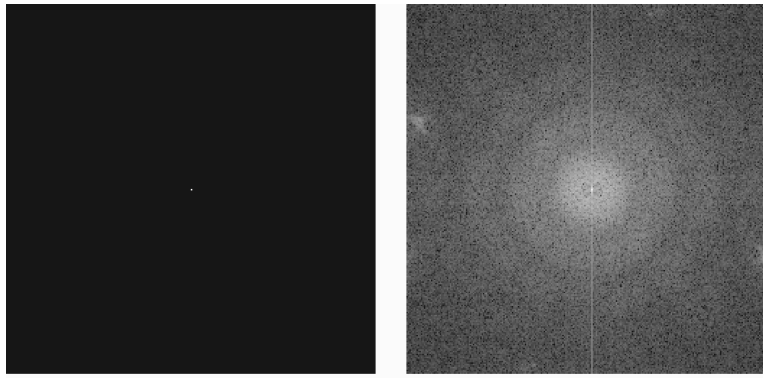

FIGURE 8.10 Example of using log transformation: (a) Fourier spectrum (amplitude only) of the rice image (available in MATLAB); (b) result of applying equation (8.6) with $c = 1$ followed by autocontrast.

In MATLAB

The `imadjust` function in the IPT can be used to perform gamma correction with the syntax: `g = imadjust(f,[],[],gamma);`

8.3.4 Log Transformations

The log transformation and its inverse are nonlinear transformations used, respectively, when we want to compress or expand the dynamic range of pixel values in an image.

Log transformations can be mathematically described as

$$s = c \cdot \log(1 + r) \tag{8.6}$$

where r is the original pixel value, s is the resulting pixel value, and c is a constant.

Be aware that in many applications of the log transformation, the input "image" is actually a 2D array with values that might lie outside the usual range for gray levels that we usually associate with monochrome images (e.g., [0, 255]).

■ EXAMPLE 8.3

This example uses the log transformation to improve the visualization and display of Fourier transform (FT)[4] results (Figure 8.10). The range of values in the matrix in part (a) is $[0, 2.8591 \times 10^4]$, which—when displayed on a linearly scaled 8-bit system—makes it hard to see anything but the bright spot at the center. Applying a log transform, the dynamic range is compressed to [0, 10.26]. Using the proper

[4]The Fourier transform (FT) and its applications will be presented in Chapter 11.

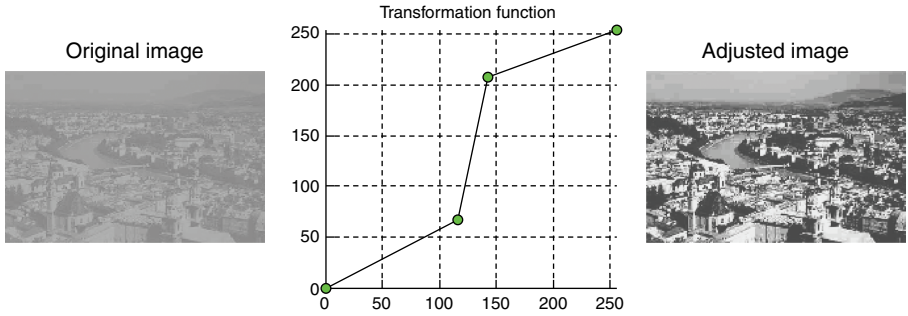

FIGURE 8.11 Piecewise linear transformation using `glsdemo`.

autocontrast transformation to linearly extend the compressed range to [0, 255], we obtain the image in part (b), where significant additional details (e.g., thin vertical line at the center, concentric circles) become noticeable.

8.3.5 Piecewise Linear Transformations

Piecewise linear transformations can be described by several linear equations, one for each interval of gray-level values in the input image. The main advantage of piecewise linear functions is that they can be arbitrarily complex; the main disadvantage is that they require additional user input [GW08], as discussed further in Section 8.4.

■ **EXAMPLE 8.4**

Figure 8.11 shows an example of an arbitrary piecewise linear transformation function used to improve the contrast of the input image. The function is specified interactively using a GUI-based MATLAB tool `glsdemo` (developed by Jeremy Jacob and available at the book's companion web site).

Figure 8.12 shows an example of *gray-level slicing*, a particular case of piecewise linear transformation in which a specific range of intensity levels (in this case, the

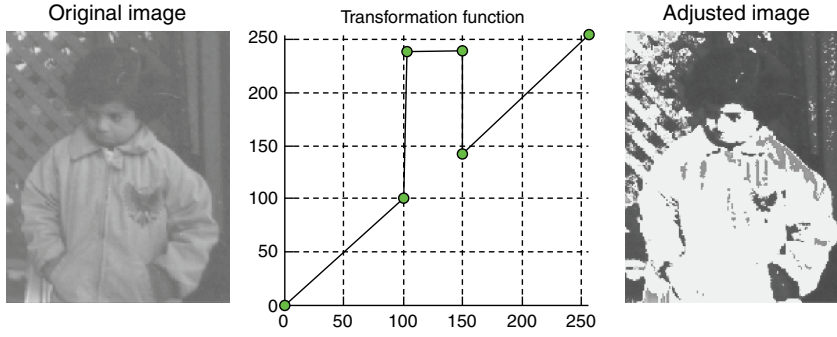

FIGURE 8.12 Gray-level slicing using `glsdemo`. Original image: courtesy of MathWorks.

[100, 150] range) is highlighted in the output image, while all other values remain untouched.[5]

8.4 SPECIFYING THE TRANSFORMATION FUNCTION

All the transformation functions presented in this chapter have been described mathematically in a way that is elegant and appropriate for input variables in the continuous or discrete domain.

In this section, we show that in spite of its elegance, the mathematical formulation is not always useful in practice, for two different—but relevant—reasons:

1. From a user interaction viewpoint, it is often preferable to specify the desired point transformation function interactively using the mouse and a GUI-based application, such as the `glsdemo` in Tutorial 8.1 (page 163).
2. From the perspective of computational efficiency, point operations can be executed at significantly higher speed using lookup tables (LUTs). For images of type `uint8` (i.e., monochrome images with 256 gray levels), the LUT will consist of a 1D array of length 256. LUTs can be easily implemented in MATLAB as demonstrated in the following examples.

■ EXAMPLE 8.5

This example shows how the piecewise linear transformation function specified by equation (8.7) can be implemented using a LUT (and the `intlut` function) in MATLAB. It is important to remember that while arrays (such as the LUT) are 1-based, pixel values vary between 0 and 255. Our MATLAB code must, therefore, include the proper adjustments to avoid off-by-one errors and attempts to access out-of-bounds array elements.

$$s = \begin{cases} 2 \cdot f & \text{for } 0 < r \leq 64 \\ 128 & \text{for } 64 < r \leq 128 \\ f & \text{for } r > 128 \end{cases} \tag{8.7}$$

Using the colon notation (see Chapter 5), we will specify the three linear portions of the transformation function as follows:

```
LUT = uint8(zeros([1 256]));
LUT(1:65) = 2*(0:64);
LUT(66:129) = 128;
LUT(130:256) = (130:256)-1;
```

[5] A variant of the gray-level slicing technique highlights a range of values and maps all other values to a fixed—and usually low—gray level.

Next, we will test the LUT using a 3×3 test image.

```
A = uint8([20 40 0; 178 198 64; 77 128 1])
B = intlut(A, LUT)
```

As expected, if the input array is

$$\begin{bmatrix} 20 & 40 & 0 \\ 178 & 198 & 64 \\ 77 & 128 & 1 \end{bmatrix}$$

the resulting array will be

$$\begin{bmatrix} 40 & 80 & 0 \\ 178 & 198 & 128 \\ 128 & 128 & 2 \end{bmatrix}$$

Finally, we will apply the LUT to a real gray-level image. The result appears in Figure 8.13. Note how many pixels in the original image have been "flattened" to the average gray level (128) in the output image.

```
I = imread('klcc_gray.png');
O = intlut(I,LUT);
figure, subplot(1,2,1), imshow(I), subplot(1,2,2), imshow(O)
```

MATLAB users should be aware that—even though MATLAB makes it extremely simple to apply a transformation function to all pixels using a single line of code—the use of a precomputed LUT enables a much more computationally efficient implementation of point transformations, as demonstrated in the following example.

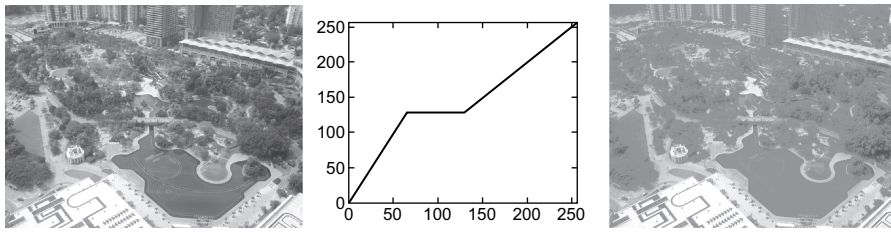

FIGURE 8.13 Example of using a lookup table: (a) input image; (b) transformation function specified by equation (8.7); (c) output image.

■ **EXAMPLE 8.6**

Let us assume that we want to implement the nonlinear transformation function $s = c\sqrt{r}$, where $c = 5$.

We will first code it using direct implementation and compute the execution times using a 1944 × 2592 pixel monochrome image as input.

```
I = imread('klcc_gray.png');
I2 = double(I);
tic
J = 5*sqrt(I2);
toc
O = uint8(J);
subplot(1,2,1), imshow(I), subplot(1,2,2), imshow(O)
```

Using MATLAB's `tic` and `toc` functions, we can measure the execution time for the transformation step. The value measured using an Apple MacBook with a 2 GHz Intel Core Duo CPU and 1 GB of RAM running MATLAB 7.6.0 (R2008a) was 0.233381 s.

Now we will repeat the process using a precomputed LUT.

```
I = imread('klcc_gray.png');
LUT = double(zeros([1 256]));
LUT(1:256) = 5 * sqrt(0:255);
LUT_int8 = uint8(LUT);
tic
O = intlut(I, LUT_int8);
toc
figure, subplot(1,2,1), imshow(I), subplot(1,2,2), imshow(O)
```

In this case, the measured execution time was 0.030049 s. Even if we move `tic` to the line immediately after the `imread` operation (i.e., if we include the time spent computing the LUT itself), the result would be 0.037198 s, which is more than six times faster than the pixel-by-pixel computation performed earlier.

8.5 TUTORIAL 8.1: GRAY-LEVEL TRANSFORMATIONS

Goal

The goal of this tutorial is to learn how to perform basic point transformations on grayscale images.

Objectives

- Explore linear transformations including the identity function and the negative function.
- Learn how to perform logarithmic grayscale transformations.
- Learn how to perform power law (gamma) grayscale transformations.
- Explore gray (intensity)-level slicing.

What You Will Need

- `radio.tif` image
- `micro.tif` image
- `glsdemo.m` script

Procedure

The most basic transformation function is the identity function, which simply maps each pixel value to the same value.

1. Create an identity transformation function.

```
x = uint8(0:255);
plot(x); xlim([0 255]); ylim([0 255]);
```

2. Use the transformation function on the `moon` image to see how the identity function works.

```
I = imread('moon.tif');
I_adj = x(I + 1);
figure, subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(I_adj), title('Adjusted Image');
```

Question 1 Why were we required to use `I+1` when performing the transformation instead of just `I`?

Question 2 How can we show that the adjusted image and the original image are equivalent?

The negative transformation function generates the negative of an image.

3. Create a negative transformation function and show the result after applied to the `moon` image.

```
y = uint8(255:-1:0); I_neg = y(I + 1);
figure, subplot(1,3,1), plot(y), ...
```

```
title('Transformation Function'), xlim([0 255]), ylim([0 255]);
subplot(1,3,2), imshow(I), title('Original Image');
subplot(1,3,3), imshow(I_neg), title('Negative Image');
```

Question 3 How did we create the negative transformation function?

A negative transformation results in the same image as if we were to complement the image logically.

4. Complement the original image and show that it is equivalent to the negative image generated in the previous step.

```
I_cmp = imcomplement(I);
I_dif = imabsdiff(I_cmp,I_neg);
figure, imshow(I_cmp)
figure, imshow(I_dif,[])
```

Logarithmic transformation functions can be used to compress the dynamic range of an image in order to bring out features that were not originally as clear. The log transformation function can be calculated using equation (8.6).

In our case, `x` represents the value of any particular pixel, and the constant `c` is used to scale the output within grayscale range [0, 255].

5. Close all open figures and clear all workspace variables.
6. Generate a logarithmic transformation function.

```
x = 0:255; c = 255 / log(256);
y = c * log(x + 1);
figure,  subplot(2,2,1), plot(y), ...
    title('Log Mapping Function'), axis tight, axis square
```

7. Use the transformation function to generate the adjusted image.

```
I = imread('radio.tif');
I_log = uint8(y(I + 1));
subplot(2,2,2), imshow(I), title('Original Image');
subplot(2,2,3), imshow(I_log), title('Adjusted Image');
```

In the second line of code, you will note that we convert the image to `uint8`. This is necessary because the only way the function `imshow` will recognize a matrix with a range of [0, 255] as an image is if it is of class `uint8`. In the next step, we will see that simply brightening the image will *not* show the missing detail in the image.

8. Show a brightened version of the image.

```
I_br = imadd(I,100);
subplot(2,2,4), imshow(I_br), title('Original Image Scaled');
```

Question 4 Why does the log-transformed image display the hidden detail in the radio image, but the brightened image does not?

The inverse of the log function is as follows.

```
y(x) = exp(x/c) - 1;
```

Again, here `c` is the scaling constant and `x` is the pixel value. We can demonstrate that applying this equation to the image we previously created (image `I_log`, which was transformed using the log transformation) will result in the original image.

9. Use inverse log transformation to undo our previous transformation.

```
%
z = exp(x/c) - 1;
I_invlog = uint8(z(I_log + 1));
figure, subplot(2,1,1), plot(z), title('Inverse-log Mapping Function');
subplot(2,1,2), imshow(I_invlog), title('Adjusted Image');
```

Power law transformations include *n*th root and *n*th power mapping functions. These functions are more versatile than the log transformation functions because you can specify the value of *n*, which ultimately changes the shape of the curve to meet your particular needs.

10. Close all open figures and clear all workspace variables.
11. Generate an *n*th root function where `n` equals 2.

```
x = 0:255; n = 2; c = 255 / (255 ^ n);
root = nthroot((x/c), n);
figure,  subplot(2,2,1), plot(root), ...
    title('2nd-root transformation'), axis tight, axis square
```

Question 5 How does the shape of the curve change if we were to use a different value for `n`?

12. Use the transformation function to generate the adjusted image.

```
I = imread('drill.tif');
I_root = uint8(root(I + 1));
subplot(2,2,2), imshow(I), title('Original Image');
subplot(2,2,[3 4]), imshow(I_root), title('Nth Root Image');
```

We can see that the adjusted image shows details that were not visible in the original image.

The *n*th power transformation function is the inverse of the *n*th root.

13. Generate an *n*th power transformation function.

```
power = c * (x .^ n);
figure, subplot(1,2,1), plot(power), ...
    title('2nd-power transformation');
axis tight, axis square
```

14. Use the *n*th power transformation to undo our previous transformation.

```
I_power = uint8(power(I_root + 1));
subplot(1,2,2), imshow(I_power), title('Adjusted Image');
```

Question 6 Show that the `I_power` image and the original image `I` are (almost) identical.

The transformation functions we explored thus far have been defined by mathematical equations. During the next steps, we shall explore the creation and application of piecewise linear transformation functions to perform specific tasks. Our first example will be *gray-level slicing*, a process by which we can enhance a particular range of the gray scale for further analysis.

15. Close all open figures and clear all workspace variables.
16. Load the `micro` image and display it.

```
I = imread('micro.tif');
figure, subplot(1,3,1), imshow(I), title('Original Image');
```

17. Create the transformation function.

```
y(1:175) = 0:174;
y(176:200) = 255;
y(201:256) = 200:255;
subplot(1,3,2), plot(y), axis tight, axis square
```

Question 7 Based on the previous step, what do you expect to be the visual effect of applying this transformation function to the original image?

18. Generate the adjusted image.

```
I2 = uint8(y(I + 1));
subplot(1,3,3), imshow(I2), title('Adjusted Image');
```

In the previous steps, we enhanced a particular range of grayscale values while leaving the others unchanged. The adjusted image reflects this change, but it is still difficult to see the enhanced pixels because of the surrounding distractions. We can isolate the enhanced pixels even more by setting all nonenhanced pixels to a constant level.

19. Create a new transformation function and display the adjusted image.

```
z(1:175) = 50;
z(176:200) = 250;
z(201:256) = 50;
I3 = uint8(z(I + 1));
figure, subplot(1,2,1), plot(z), ...
    xlim([0 255]), ylim([0 255]), axis square
subplot(1,2,2), imshow(I3)
```

Although it is possible to create any transformation function in MATLAB by defining a vector of values, as we did above, this can be tedious and time consuming. Through the use of a GUI, we can dynamically generate a function that the user defines visually. The following demo illustrates this.

20. Review the help information for `glsdemo`.
21. Run `glsdemo` with the image `micro.tif` and recreate the transformation functions that we previously used in steps 17 and 19.

WHAT HAVE WE LEARNED?

- Image enhancement is the process of modifying the pixel values within an image in such a way that the resulting image is an improved version of the original image for a particular purpose, whether it is the human perception of subjective quality or further processing by machine vision algorithms.
- Image enhancement can be achieved in many different ways, including the use of certain gray-level transformations, whose chief characteristic is the fact that the resulting gray level of a pixel depends only on the original pixel value and the transformation function. For this reason, gray-level transformations are also referred to as *point transformations*.
- Gray-level transformations can be implemented in MATLAB using the `imadjust` function.
- Gray-level transformations are often used for brightness and contrast adjustments. In MATLAB, interactive brightness and contrast adjustments can also be performed using `imcontrast`.
- Some of the most commonly used point transformations are negative, power law (gamma), logarithmic, and piecewise linear transformations.

- Point transformation functions can be specified interactively using tools such as `glsdemo`.
- The use of a lookup table speeds up the processing of point transformation functions.

LEARN MORE ABOUT IT

- Sections 10.1–10.3 of [Jah05] discuss point transformations and applications such as noise variance equalization, two-point radiometric calibration, and windowing.
- Section 5.4 of [BB08] presents a modified autocontrast operation.
- Section 3.2 of [GWE04] contains MATLAB functions (`intrans` and `gscale`) that extend the functionality of `imadjust`.
- Section 3.8 of [GW08] discusses the use of fuzzy techniques for intensity transformations.

8.6 PROBLEMS

8.1 Write a MATLAB function to perform a piecewise linear brightness and contrast adjustment on monochrome images using the generic method described in equation (8.3). It should take as arguments a monochrome image, the c coefficient (slope), and the b coefficient (offset).

8.2 Write a MATLAB function to perform a simple version of *image solarization* technique (also known as *Sabatier effect*), a point transformation that processes an image by leaving all pixels brighter than a certain value (T) untouched, while extracting the negative of all pixels darker than T.

8.3 Write a MATLAB function to perform a point transformation by which each pixel value in an input image of class `uint8` is replaced by the square of its original value and answer the following questions:

1. Do you have to explicitly make provisions for clamping the results (so that they stay within range)? Why (not)?
2. Is the resulting image brighter or darker than the original image? Explain.

8.4 Repeat Problem 8.3, this time for an input image of class `double`. Does the answer to any of the questions change? Why?

8.5 The sigmoid function used to generate a point transformation similar to Figure 8.5a can be described by the equation

$$s = \frac{1}{1 + (m/r)^S} \tag{8.8}$$

where r is the original pixel value, s is the resulting pixel value, m is a user-specified threshold, and S is a parameter that controls the slope of the curve.

Write a MATLAB script that generates and plots the point transformation function described in equation (8.8).

8.6 Apply the transformation function developed for Problem 8.5 to different images and experiment with different values of m and S. Report a summary of your findings.