FPT Education

FPT UNIVERSITY

PART I
IMAGE PROCESSING

INSTRUCTOR: DR. NGUYEN NGOC TRUONG MINH

SCHOOL OF ELECTRICAL ENGINEERING, INTERNATIONAL UNIVERSITY (VNU-HCMC)

Ho Chi Minh City, June 2023

LECTURE V – ARITHMETIC AND LOGIC, GEOMETRICS OPERATIONS

INSTRUCTOR: DR. NGUYEN NGOC TRUONG MINH

SCHOOL OF ELECTRICAL ENGINEERING, INTERNATIONAL UNIVERSITY (VNU-HCMC)

Ho Chi Minh City, June 2023

LECTURE CONTENT

- Which arithmetic and logic operations can be applied to digital images?
- How are they performed in MATLAB?
- What are they used for?
- What do geometric operations do to an image and what are they used for?
- What are the techniques used to enlarge/reduce a digital image?
- What are the main interpolation methods used in association with geometric operations?

LECTURE CONTENT

- What are affine transformations and how can they be performed using MATLAB?
- How can I rotate, flip, crop, or resize images in MATLAB?
- What is image registration and where is it used?
- Chapter Summary
- Problems

5.1 ARITHMETIC OPERATIONS: FUNDAMENTALS AND APPLICATIONS

- Arithmetic operations involving images are typically performed on *a pixel-by-pixel basis*; that is, the operation is independently applied to each pixel in the image. Given a 2D array (X) and another 2D array of the same size or a scalar (Y), the resulting array, Z, is obtained by calculating

$$X \ opn \ Y = Z$$

where opn is a binary arithmetic (+, −, ×, /) operator.

- This section describes each arithmetic operation in more detail, focusing on how they can be performed and what are their typical applications.

5.1.1 ADDITION

- Addition is used to blend the pixel contents from two images or add a constant value to pixel values of an image. Adding the contents of two monochrome images causes their contents to blend (Figure 6.1).

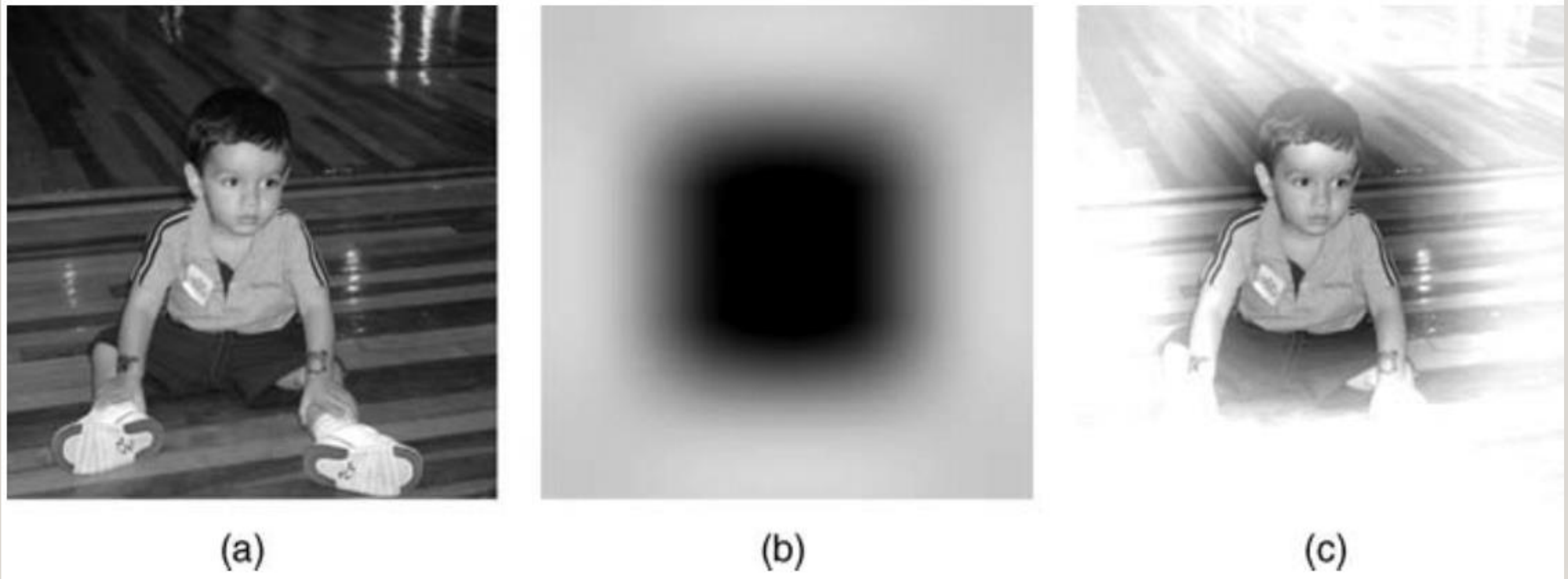


FIGURE 6.1 Adding two images: (a) first image (X); (b) second image (Y); (c) result ($Z = X + Y$).

5.1.1 ADDITION

- Adding a constant value (scalar) to an image causes an increase (or decrease if the value is less than zero) in its overall brightness, a process sometimes referred to as additive image offset (Figure 6.2).

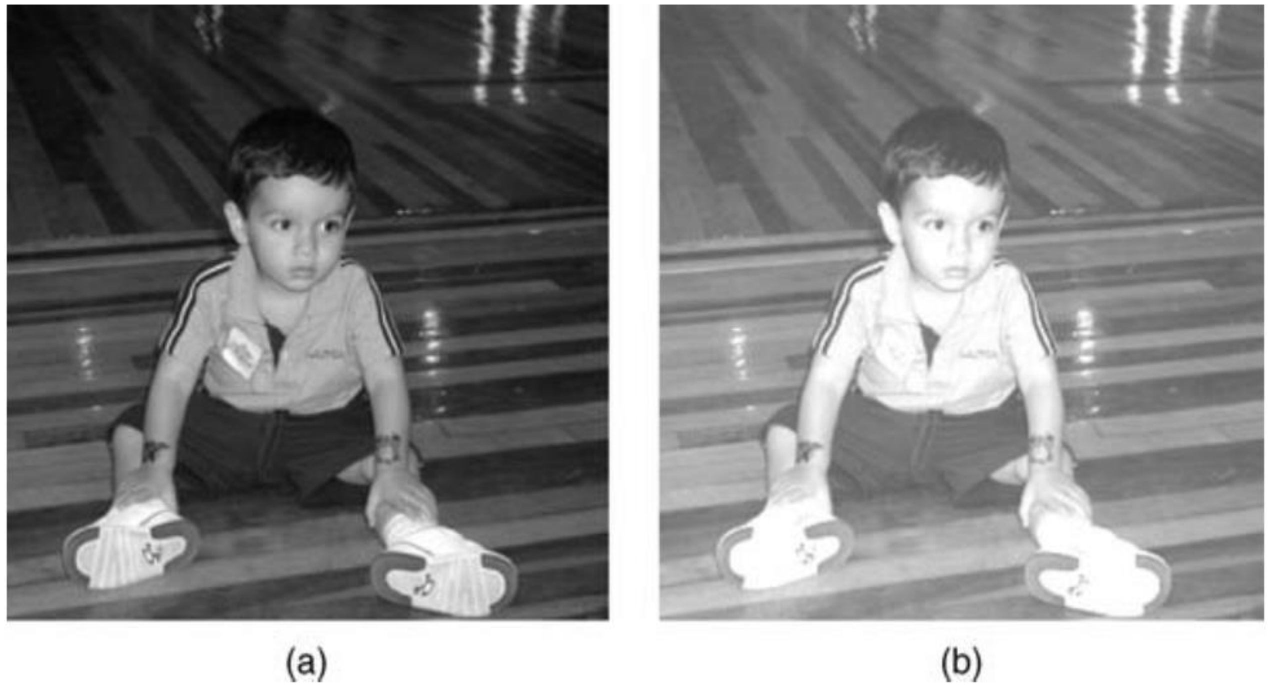


FIGURE 6.2 Additive image offset: (a) original image (X); (b) brighter version ($Z = X + 75$).

5.1.1 ADDITION

- Adding random amounts to each pixel value is a common way to simulate additive noise (Figure 6.3). The resulting (noisy) image is typically used as a test image for restoration algorithms.

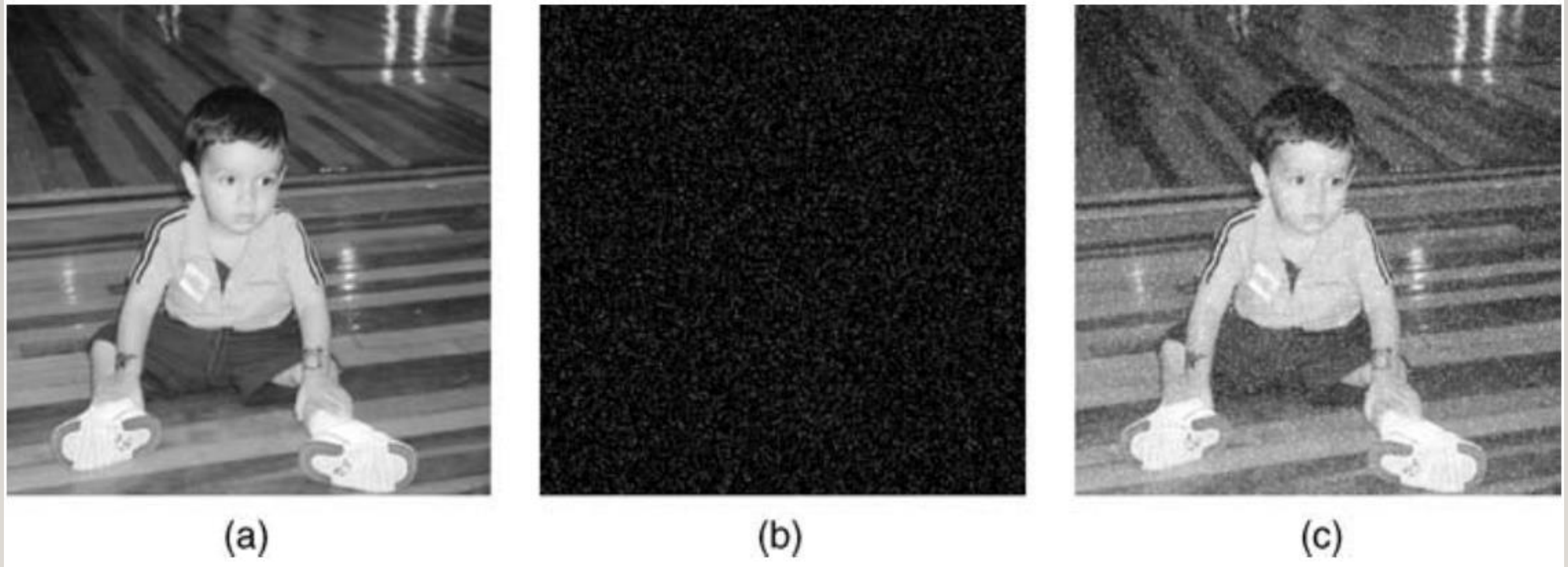


FIGURE 6.3 Adding noise to an image: (a) original image (X); (b) zero-mean Gaussian white noise (variance = 0.01) (N); (c) result ($Z = X + N$).

5.1.1 ADDITION

In MATLAB

- MATLAB's IPT has a built-in function to add two images or add a constant (scalar) to an image: *imadd*. When adding two images, you must be careful with values that exceed the maximum pixel value for the data type being used. There are two ways of dealing with this overflow issue: *normalization* and *truncation*.

5.1.1 ADDITION

In MATLAB

- *Normalization* consists in storing the intermediate result in a temporary variable (W) and calculating each resulting pixel value in Z using:

$$g = \frac{L_{max}}{f_{max} - f_{min}}(f - f_{min})$$

where f is the current pixel in W, L_{max} is the maximum possible intensity value (e.g., 255 for *uint8* or 1.0 for *double*), g is the corresponding pixel in Z, f_{max} is the maximum pixel value in W, and f_{min} is the minimum pixel value in W.

- *Truncation* consists in simply limiting the results to the maximum positive number that can be represented with the adopted data type.

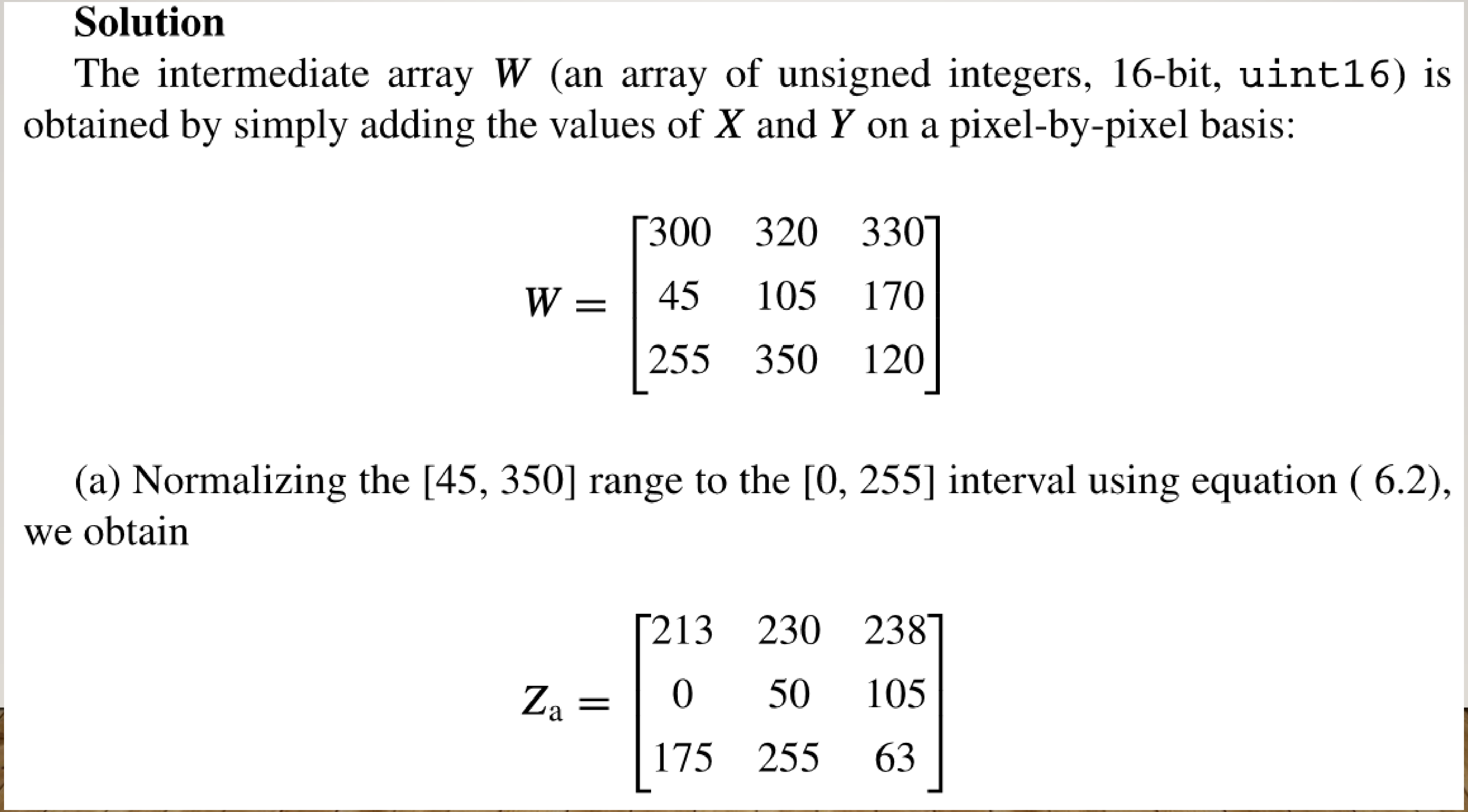
EXAMPLE 5.1

- For the two 3 x 3 monochrome images below (X and Y), each of which represented as an array of unsigned integers, 8-bit (*uint8*), calculate $Z = X + Y$, using (a) *normalization* and (b) *truncation*.

$$X = \begin{bmatrix} 200 & 100 & 100 \\ 0 & 10 & 50 \\ 50 & 250 & 120 \end{bmatrix}$$

$$Y = \begin{bmatrix} 100 & 220 & 230 \\ 45 & 95 & 120 \\ 205 & 100 & 0 \end{bmatrix}$$

EXAMPLE 5.1



Solution

The intermediate array W (an array of unsigned integers, 16-bit, `uint16`) is obtained by simply adding the values of X and Y on a pixel-by-pixel basis:

$$W = \begin{bmatrix} 300 & 320 & 330 \\ 45 & 105 & 170 \\ 255 & 350 & 120 \end{bmatrix}$$

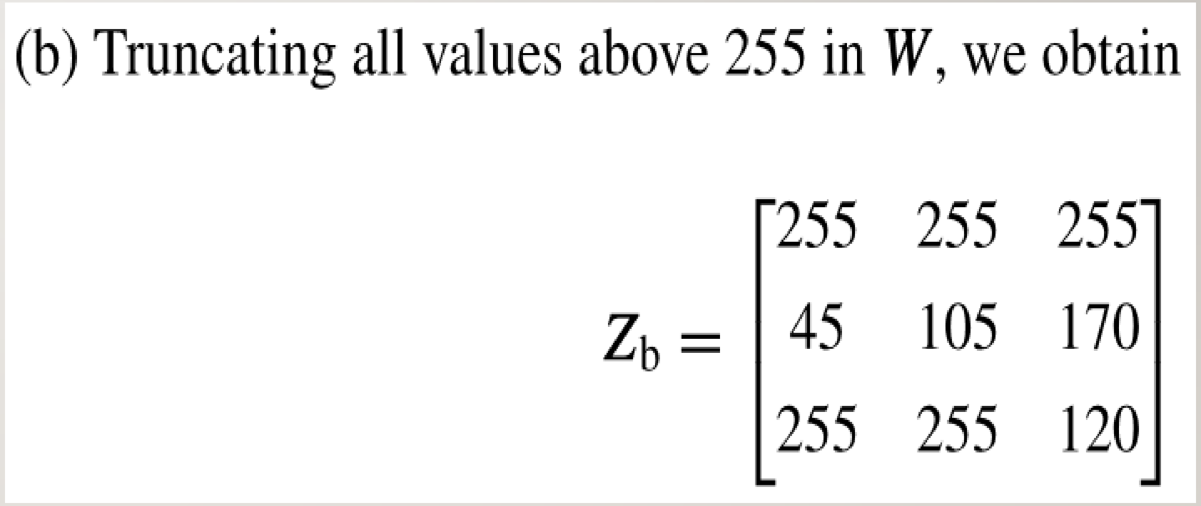
(a) Normalizing the [45, 350] range to the [0, 255] interval using equation (6.2), we obtain

$$Z_{\text{a}} = \begin{bmatrix} 213 & 230 & 238 \\ 0 & 50 & 105 \\ 175 & 255 & 63 \end{bmatrix}$$

EXAMPLE 5.1

MATLAB code

```
X = uint8([200 100 100; 0 10 50; 50 250 120]);
Y = uint8([100 220 230; 45 95 120; 205 100 0]);
W = uint16(X) + uint16(Y);
fmax = max(W(:));
fmin = min(W(:));
Za = uint8(255.0*double((W-fmin))/double((fmax-fmin)));
Zb = imadd(X,Y);
```



(b) Truncating all values above 255 in W, we obtain

$$Z_b = \begin{bmatrix} 255 & 255 & 255 \\ 45 & 105 & 170 \\ 255 & 255 & 120 \end{bmatrix}$$

5.1.2 SUBTRACTION

- *Subtraction* is often used to *detect differences between two images*. Such differences may be due to several factors, such as artificial addition to or removal of relevant contents from the image (e.g., using an image manipulation program), relative object motion between two frames of a video sequence, and many others.

- *Subtracting a constant value (scalar) from an image causes a decrease in its overall brightness*, a process sometimes referred to as *subtractive image offset* (Figure 6.4).

5.1.2 SUBTRACTION

- When subtracting one image from another or a constant (*scalar*) from an image, we must be careful with the possibility of obtaining negative pixel values as a result.

- There are two ways of dealing with this *underflow* issue: treating subtraction as absolute difference (which will always result in positive values proportional to the difference between the two original images without indicating, however, which pixel was brighter or darker) and truncating the result, so that negative intermediate values become zero.

5.1.2 SUBTRACTION

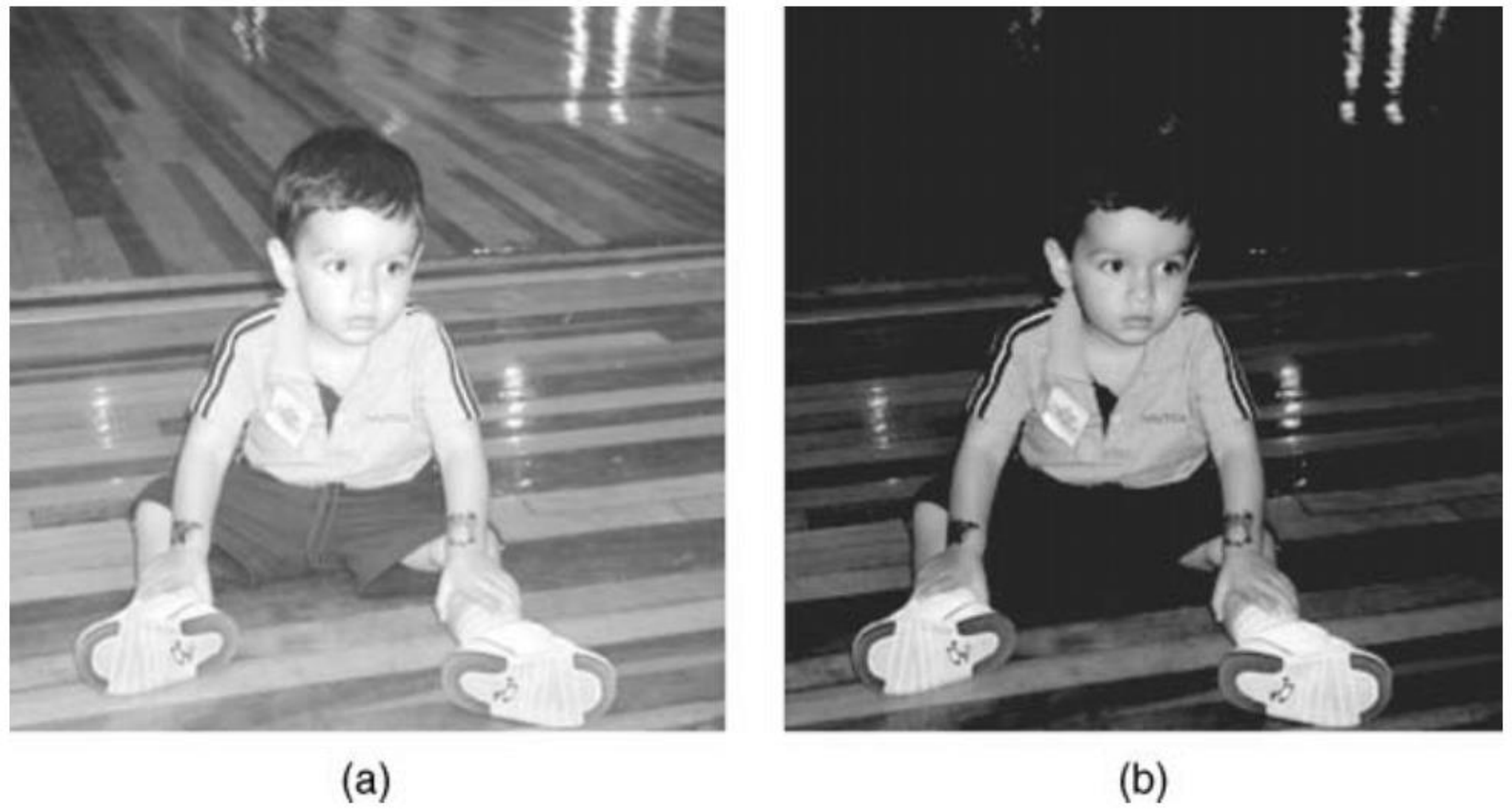


FIGURE 6.4 Subtractive image offset: (a) original image (X); (b) darker version ($Z = X - 75$).

5.1.2 SUBTRACTION

In MATLAB

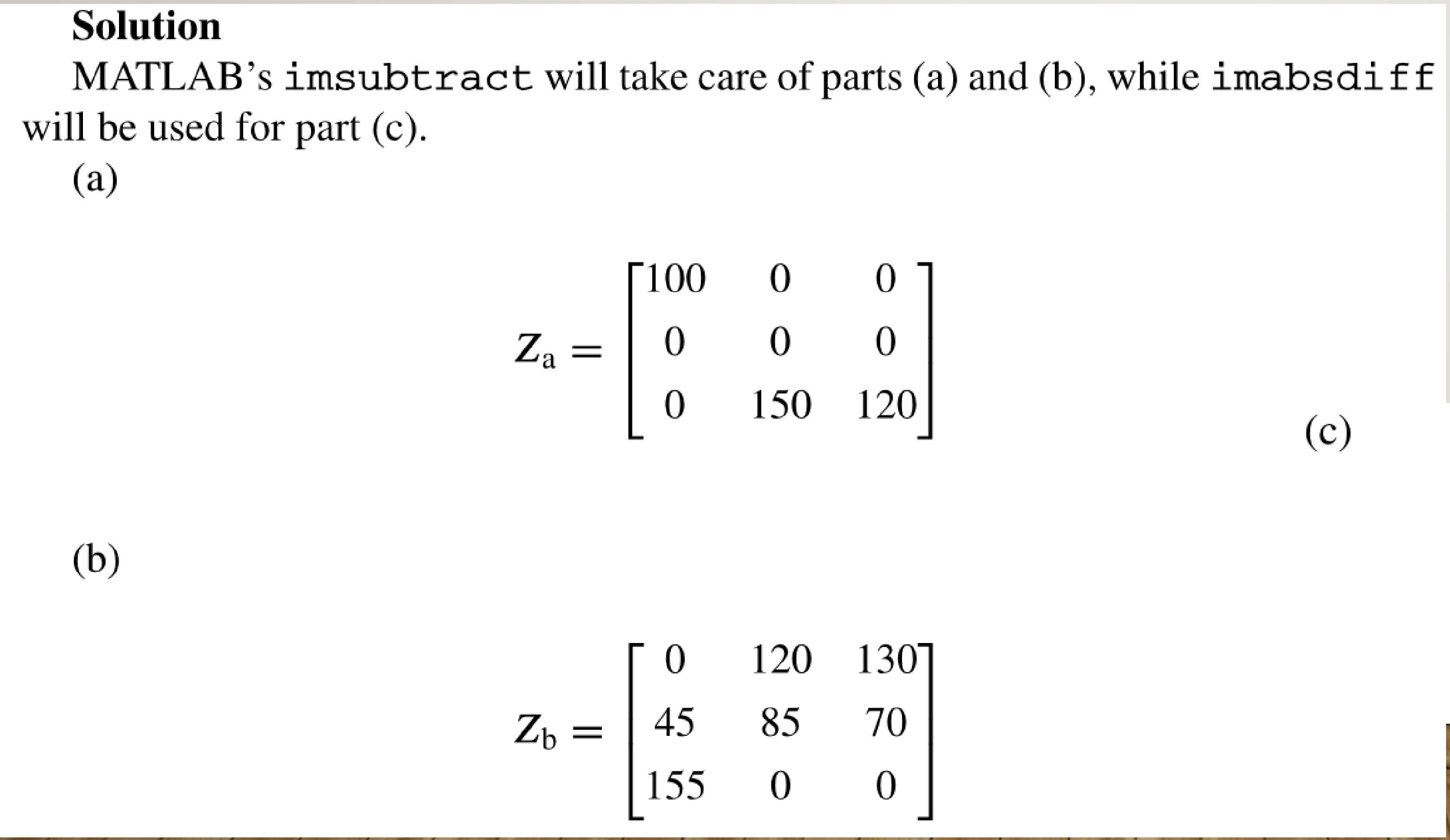
- The IPT has a built-in function to subtract one image from another or subtract a constant from an image: *imsubtract*. The IPT also has a built-in function to calculate the absolute difference of two images: *imabsdiff*. The IPT also includes a function for calculating the negative (complement) of an image, *imcomplement*.

EXAMPLE 5.2

- For the two 3 x 3 monochrome images below (X and Y), each of which represented as an array of unsigned integers, 8-bit (*uint8*), calculate (a) $Z = X - Y$, (b) $Z = Y - X$, and (c) $Z = |Y - X|$. For parts (a) and (b), use truncation to deal with possible negative values.

$$\boldsymbol{X} = \begin{bmatrix} 200 & 100 & 100 \\ 0 & 10 & 50 \\ 50 & 250 & 120 \end{bmatrix} \qquad \boldsymbol{Y} = \begin{bmatrix} 100 & 220 & 230 \\ 45 & 95 & 120 \\ 205 & 100 & 0 \end{bmatrix}$$

EXAMPLE 5.2



Solution

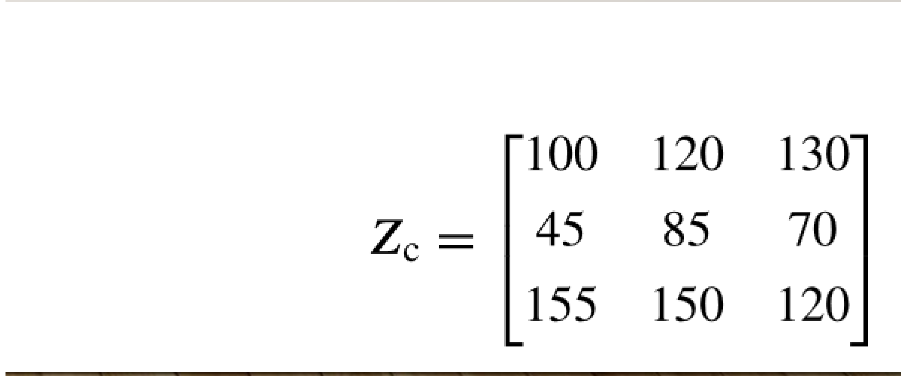
MATLAB's `imsubtract` will take care of parts (a) and (b), while `imabsdiff` will be used for part (c).

(a)

$$Z_a = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 150 & 120 \end{bmatrix}$$

(b)

$$Z_b = \begin{bmatrix} 0 & 120 & 130 \\ 45 & 85 & 70 \\ 155 & 0 & 0 \end{bmatrix}$$



(c)

$$Z_c = \begin{bmatrix} 100 & 120 & 130 \\ 45 & 85 & 70 \\ 155 & 150 & 120 \end{bmatrix}$$

EXAMPLE 5.2

MATLAB code:

```
X = uint8([200 100 100; 0 10 50; 50 250 120]);
Y = uint8([100 220 230; 45 95 120; 205 100 0]);
Za = imsubtract(X,Y);
Zb = imsubtract(Y,X);
Zc = imabsdiff(Y,X);
```

Image subtraction can also be used to obtain the negative of an image

$$g = -f + L_{max}$$

where L_{max} is the maximum possible intensity value (e.g., 255 for uint8 or 1.0 for *double*), f is the pixel value in X, g is the corresponding pixel in Z.

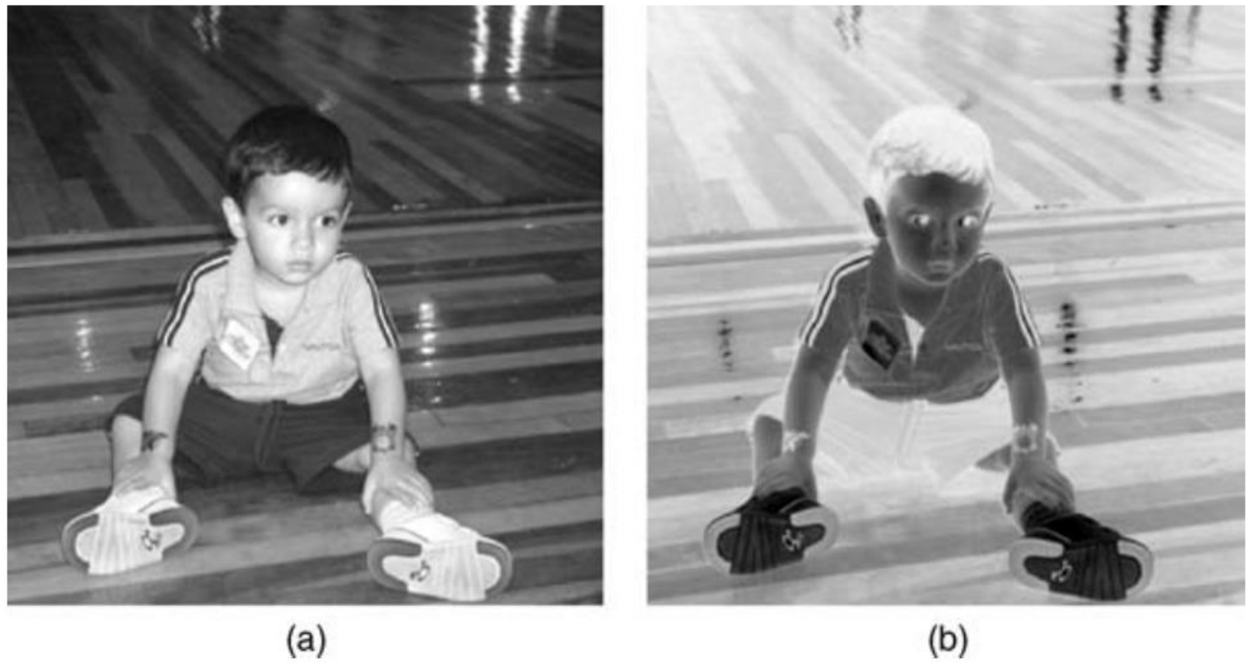


FIGURE 6.5 Example of an image negative: (a) original image; (b) negative image.

5.1.3 MULTIPLICATION AND DIVISION

- Multiplication and division by a scalar are often used to perform brightness adjustments on an image. This process—sometimes referred to as *multiplicative image scaling*—makes each pixel value brighter (or darker) by multiplying its original value by a scalar factor: *if the value of the scalar multiplication factor is greater than one, the result is a brighter image*; *if it is greater than zero and less than one, it results in a darker image* (Figure 6.6).

- Multiplicative image scaling usually produces better subjective results than the additive image offset process described previously.

5.1.3 MULTIPLICATION AND DIVISION

- The IPT has a built-in function to multiply two images or multiply an image by a constant: *immultiply*. The IPT also has a built-in function to divide one image into another or divide an image by a constant: *imdivide*.

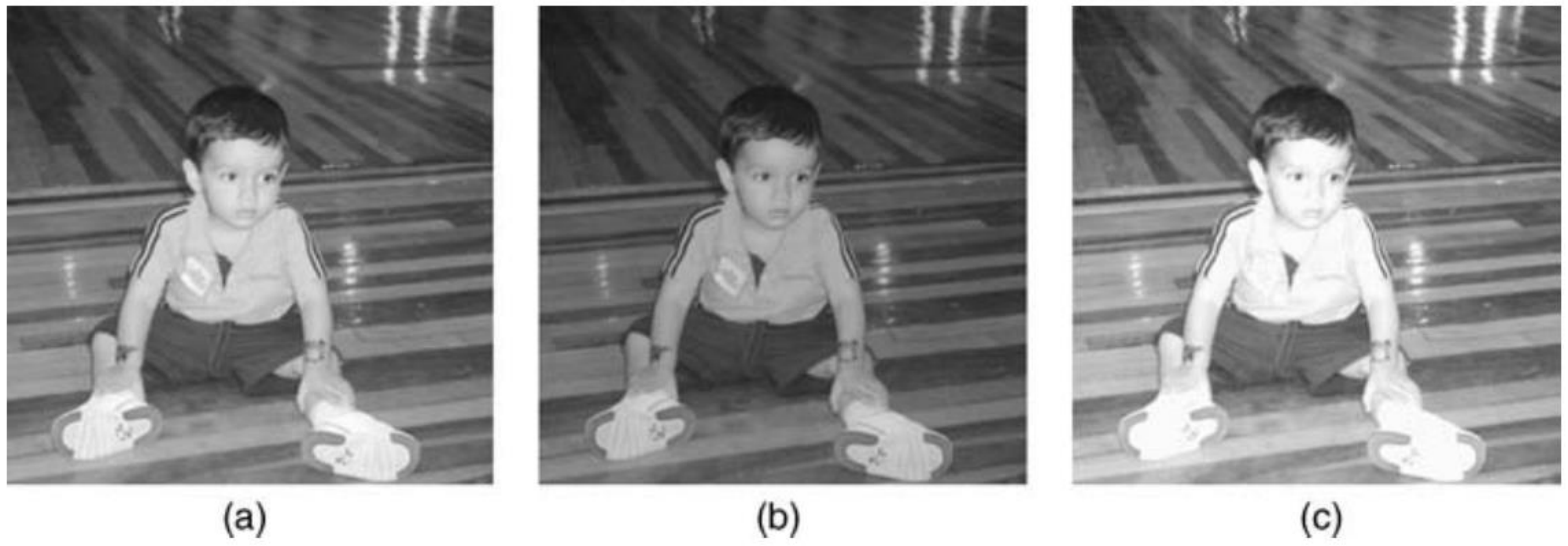


FIGURE 6.6 Multiplication and division by a constant: (a) original image (X); (b) multiplication result ($X \times 0.7$); (c) division result ($X/0.7$).

5.1.4 COMBINING SEVERAL ARITHMETIC OPERATIONS

- It is sometimes necessary to *combine several arithmetic operations applied to one or more images*, which may compound the problems of overflow and underflow discussed previously.
- To achieve more accurate results without having to explicitly handle truncations and round-offs, the IPT offers a built-in function to perform a linear combination of two or more images: *imlincomb*. This function computes each element of the output individually, in double-precision floating point.
- If the output is an integer array, *imlincomb* truncates elements that exceed the range of the integer type and rounds off fractional values.

EXAMPLE 5.3

- Calculate the average of the three 3 x 3 monochrome images below (X, Y, and Z), each of which represented as an array of unsigned integers, 8-bit (*uint8*), using (a) *imadd* and *imdivide* without explicitly handling truncation and round-offs; (b) *imadd* and *imdivide*, but this time handling truncation and round-offs; and (c) *imlincomb*.

$$X = \begin{bmatrix} 200 & 100 & 100 \\ 0 & 10 & 50 \\ 50 & 250 & 120 \end{bmatrix} \quad Y = \begin{bmatrix} 100 & 220 & 230 \\ 45 & 95 & 120 \\ 205 & 100 & 0 \end{bmatrix} \quad Z = \begin{bmatrix} 200 & 160 & 130 \\ 145 & 195 & 120 \\ 105 & 240 & 150 \end{bmatrix}$$

EXAMPLE 5.3

Solution

(a)

$$S_a = \begin{bmatrix} 85 & 85 & 85 \\ 63 & 85 & 85 \\ 85 & 85 & 85 \end{bmatrix}$$

(b)

$$S_b = \begin{bmatrix} 167 & 160 & 153 \\ 63 & 100 & 97 \\ 120 & 197 & 90 \end{bmatrix}$$

(c)

$$S_c = \begin{bmatrix} 167 & 160 & 153 \\ 63 & 100 & 97 \\ 120 & 197 & 90 \end{bmatrix}$$

EXAMPLE 5.3

MATLAB code:

```
X = uint8([200 100 100; 0 10 50; 50 250 120]);
Y = uint8([100 220 230; 45 95 120; 205 100 0]);
Z = uint8([200 160 130; 145 195 120; 105 240 150]);
Sa = imdivide(imadd(X,imadd(Y,Z)),3);
a = uint16(X) + uint16(Y);
B = a+uint16(Z);
Sb = uint8(b/3);
Sc = imlincomb(1/3,X,1/3,Y,1/3,Z,'uint8');
```

The result in (a) is incorrect due to truncation of intermediate results. Both (b) and (c) produce correct results, but the solution using *imlincomb* is much more elegant and concise.

5.2 LOGIC OPERATIONS: FUNDAMENTALS AND APPLICATIONS

- Logic operations are performed in *a bit-wise fashion on the binary contents of each pixel value.* The AND, XOR, and OR operators require two or more arguments, whereas the NOT operator requires only one argument.
- Figure 6.7 shows the most common logic operations applied to binary images, using the following convention: 1 (true) for white pixels and 0 (false) for black pixels.

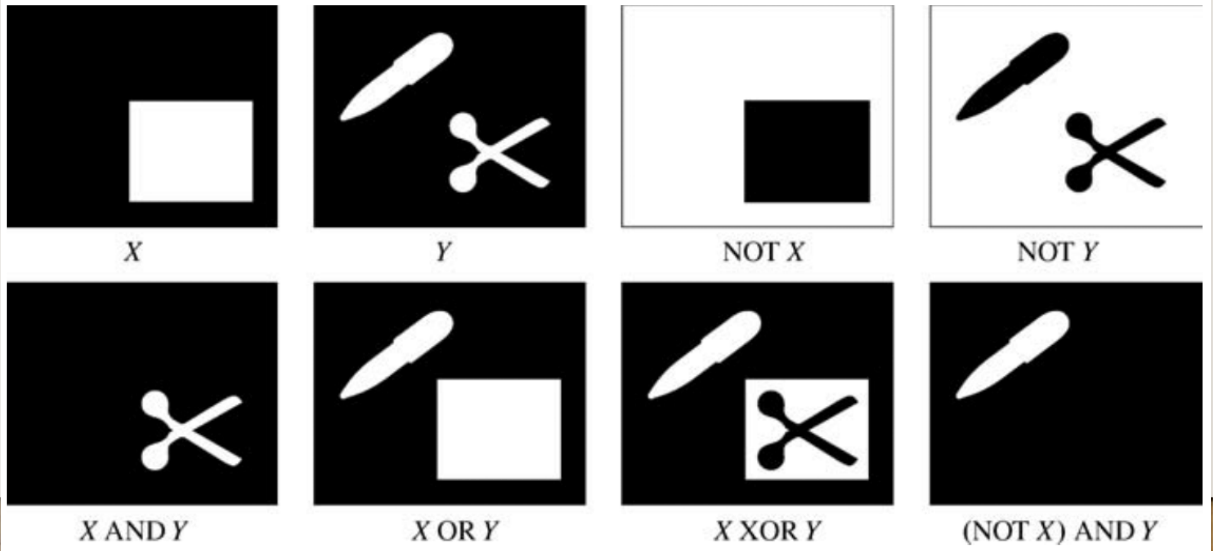


FIGURE 6.7 Logic operations on binary images.

In MATLAB

MATLAB has built-in functions to perform logic operations on arrays: *bitand*, *bitor*, *bitxor*, and *bitcmp*.

5.2 LOGIC OPERATIONS: FUNDAMENTALS AND APPLICATIONS

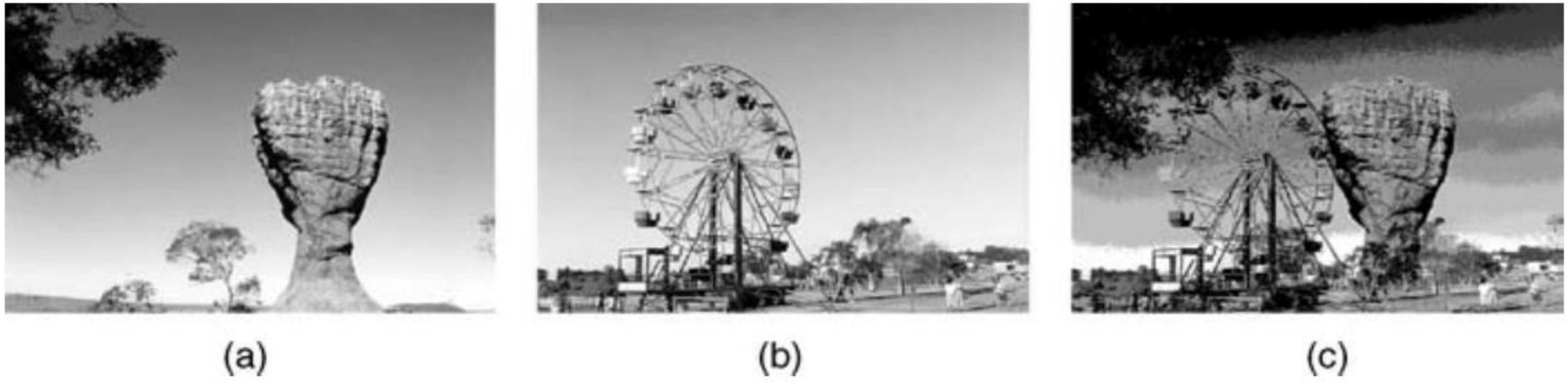


FIGURE 6.8 The AND operation applied to monochrome images: (a) X; (b) Y; (c) X AND Y.

5.2 LOGIC OPERATIONS: FUNDAMENTALS AND APPLICATIONS

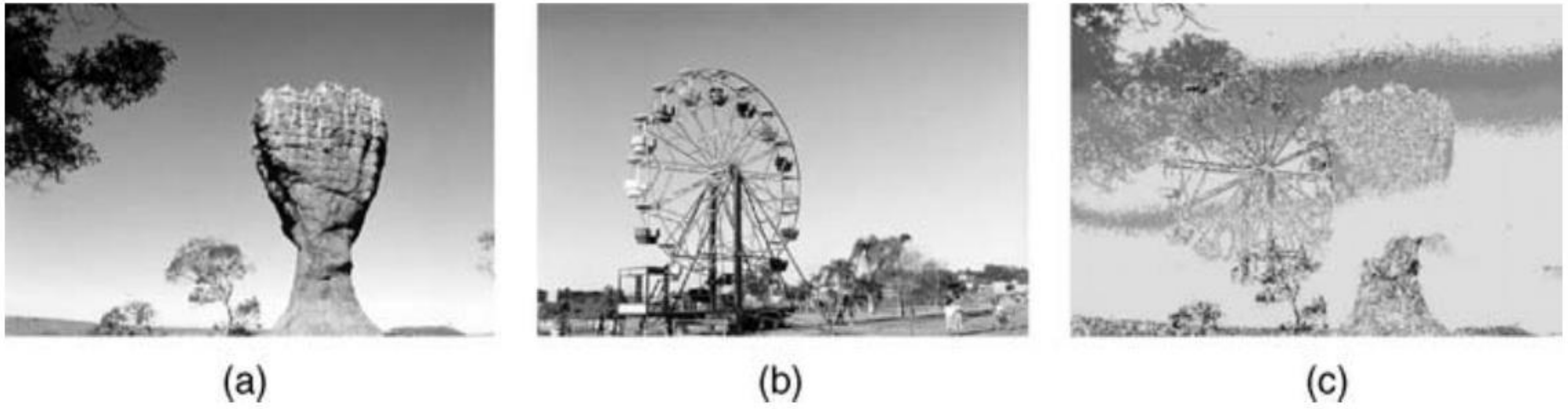


FIGURE 6.9 The OR operation applied to monochrome images: (a) X; (b) Y; (c) X OR Y.

5.2 LOGIC OPERATIONS: FUNDAMENTALS AND APPLICATIONS

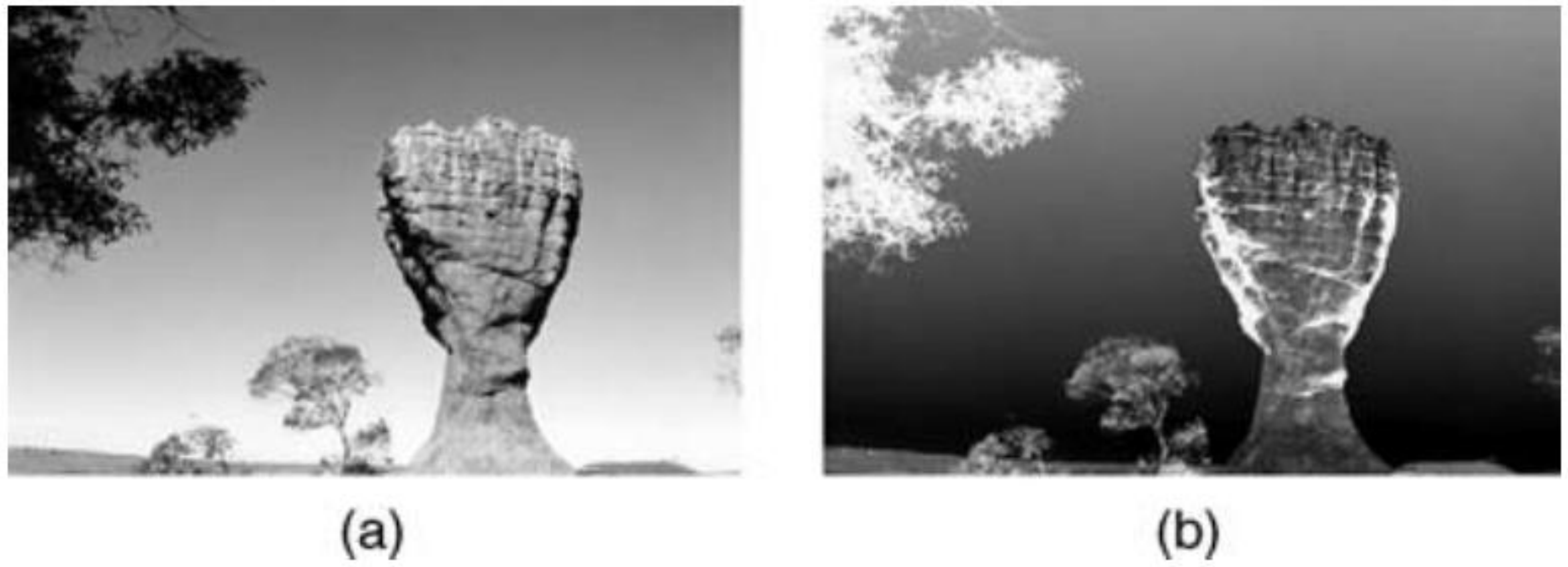


FIGURE 6.11 The NOT operation applied to a monochrome image: (a) *X*; (b) NOT *X*.

5.2 LOGIC OPERATIONS: FUNDAMENTALS AND APPLICATIONS

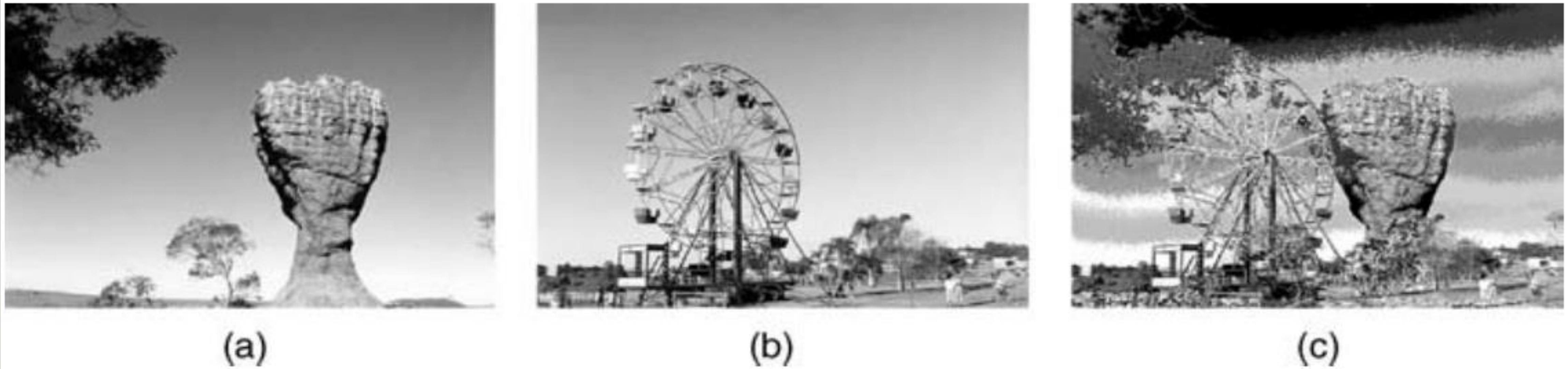


FIGURE 6.10 The XOR operation applied to monochrome images: (a) X; (b) Y; (c) X XOR Y.

5.3 TUTORIAL: ARITHMETIC OPERATIONS

Goal

The goal of this tutorial is to learn how to perform arithmetic operations on images.

Objectives

- *Learn how to perform image addition using the imadd function.*
- *Explore image subtraction using the imsubtract function.*
- *Explore image multiplication using the immultiply function.*
- *Learn how to use the imdivide function for image division.*

5.3 TUTORIAL: ARITHMETIC OPERATIONS

1. Use the imadd function to brighten an image by adding a constant (scalar) value to all its pixel values.

```
I = imread('tire.tif');
I2 = imadd(I,75);
figure;
subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(I2), title('Brighter Image');
```

Question 1: What are the maximum and minimum values of the original and the adjusted image? Explain your results.

Question 2: How many pixels had a value of 255 in the original image and how many have a value of 255 in the resulting image?

5.3 TUTORIAL: ARITHMETIC OPERATIONS

2. Use the *imadd* function to blend two images.

```
Ia = imread('rice.png');
Ib = imread('cameraman.tif');
Ic = imadd(Ia,Ib);
figure;
imshow(Ic);
```

Image subtraction is useful when determining whether two images are the same. By subtracting one image from another, we can highlight the differences between the two.

3. Close all open figures and clear all workspace variables.

5.3 TUTORIAL: ARITHMETIC OPERATIONS

4. Load two images and display them.

```
I = imread('cameraman.tif');
J = imread('cameraman2.tif');
figure;
subplot(1,2,1), imshow(I), title('Original Image');
subplot(1,2,2), imshow(J), title('Altered Image');
```

While it may not be obvious at first how the altered image differs from the original image, we should be able to see where the difference is located after using the imsubtract function.

5.3 TUTORIAL: ARITHMETIC OPERATIONS

5. Subtract both images and display the result.

```
diffim = imsubtract(I,J);
figure
subplot(2,2,1), imshow(diffim), title('Subtracted Image');
```

6. Use the zoom tool to zoom into the right area of the difference image about halfway down the image. You will notice that a small region of pixels is faintly white.

7. To zoom back out, double-click anywhere on the image.

Now that we know where the difference is located, we can look at the original images to see the change. *The difference image above does not quite seem to display all the details of the missing building.*

5.3 TUTORIAL: ARITHMETIC OPERATIONS

This is because when we performed image subtraction, some of the pixels resulted in negative values, but were then set to 0 by the *imsubtract* function (the function does this on purpose to keep the data within grayscale range). What we really want to do is calculate the absolute value of the difference between two images.

8. Calculate the absolute difference. Make sure Figure 2 is selected before executing this code.

```
diffim2 = imabsdiff(I,J);
subplot(2,2,2), imshow(diffim2), title('Abs Diff Image');
```

5.3 TUTORIAL: ARITHMETIC OPERATIONS

9. Use the zoom-in tool to inspect the new difference image.

Even though the new image may look the same as the previous one, it represents both positive and negative differences between the two images. To see this difference better, we will scale both difference images for display purposes, so their values occupy the full range of the gray scale.

10. Show scaled versions of both difference images.

```
subplot(2,2,3), imshow(diffim,[]), ...
title('Subtracted Image Scaled');
```

11. Use the zoom tool to see the differences between all four difference images.

5.3 TUTORIAL: ARITHMETIC OPERATIONS

Question 3: How did we scale the image output?

Question 4: What happened when we scaled the difference images?

Question 5: Why does the last image show more detail than the others?

Multiplication is the process of multiplying the values of each pixel of same coordinates in two images. This can be used for a brightening process known as *dynamic scaling*, which results in a more naturally brighter image compared to directly adding a constant to each pixel.

12. Close all open figures and clear all workspace variables.

5.3 TUTORIAL: ARITHMETIC OPERATIONS

13. Use *immultiply* to dynamically scale the moon image.

```
I = imread('moon.tif');
I2 = imadd(I,50);
I3 = immultiply(I,1.2);
figure
subplot(1,3,1), imshow(I), title('Original Image');
subplot(1,3,2), imshow(I2), title('Normal Brightening');
subplot(1,3,3), imshow(I3), title('Dynamic Scaling');
```

Question 6: When dynamically scaling the moon image, why did the dark regions around the moon not become brighter as in the normally adjusted image?

5.3 TUTORIAL: ARITHMETIC OPERATIONS

Image multiplication can also be used for special effects such as an artificial 3D look. By multiplying a flat image with a gradient, we create the illusion of a 3D textured surface.

14. Close all open figures and clear all workspace variables.

15. Create an artificial 3D planet by using the *immultiply* function to multiply the earth1 and earth2 images.

```
I = im2double(imread('earth1.tif'));
J = im2double(imread('earth2.tif'));
K = immultiply(I,J);
figure
subplot(1,3,1), imshow(I), title('Planet Image');
subplot(1,3,2), imshow(J), title('Gradient');
subplot(1,3,3), imshow(K,[]), title('3D Planet');
```

5.3 TUTORIAL: ARITHMETIC OPERATIONS

Image division can be used as the inverse operation to dynamic scaling. Image division is accomplished with the *imdivide* function. When using image division for this purpose, we can achieve the same effect using the *immultiply* function.

16. Close all open figures and clear all workspace variables.
17. Use image division to dynamically darken the moon image.

```
I = imread('moon.tif');
I2 = imdivide(I,2);
figure
subplot(1,3,1), imshow(I), title('Original Image');
subplot(1,3,2), imshow(I2), title('Darker Image w/ Division');
```

5.3 TUTORIAL: ARITHMETIC OPERATIONS

18. Display the equivalent darker image using image multiplication.

```
I3 = immultiply(I,0.5);
subplot(1,3,3), imshow(I3), ...
title('Darker Image w/ Multiplication');
```

Question 7: Why did the multiplication procedure produce the same result as division?

Question 8: Write a small script that will verify that the images produced from division and multiplication are equivalent.

Another use of the image division process is to extract the background from an image. This is usually done during a preprocessing stage of a larger, more complex operation.

19. Close all open figures and clear all workspace variables.

5.3 TUTORIAL: ARITHMETIC OPERATIONS

20. Load the images that will be used for background subtraction.

```
notext = imread('gradient.tif');
text = imread('gradient_with_text.tif');
figure, imshow(text), title('Original Image');
```

This image could represent a document that was scanned under inconsistent lighting conditions. Because of the background, the text in this image cannot be processed directly—we must preprocess the image before we can do anything with the text. If the background were homogeneous, we could use image thresholding to extract the text pixels from the background. Thresholding is a simple process of converting an image to its binary equivalent by defining a threshold to be used as a cutoff value: anything below the threshold will be discarded (set to 0) and anything above it will be kept (set to 1 or 255, depending on the data class we choose).

5.3 TUTORIAL: ARITHMETIC OPERATIONS

21. Show how thresholding fails in this case.

```
level = graythresh(text);
BW = im2bw(text,level);
figure, imshow(BW);
```

Although the specifics of the thresholding operation (using built-in functions *graythresh* and *im2bw*) are not important at this time, we can see that even though we attempted to segregate the image into dark and light pixels, it produced only part of the text we need (on the upper right portion of the image). If an image of the background with no text on it is available, we can use the *imdivide* function to extract the letters. To obtain such background image in a real scenario, such as scanning documents, a blank page that would show only the inconsistently lit background could be scanned.

5.3 TUTORIAL: ARITHMETIC OPERATIONS

22. Divide the background from the image to get rid of the background.

```
fixed = imdivide(text,notext);
figure;
subplot(1,3,1), imshow(text), title('Original Image');
subplot(1,3,2), imshow(notext), title('Background Only');
subplot(1,3,3), imshow(fixed,[]), title('Divided Image');
```

Question 9: Would this technique still work if we were unable to obtain the background image?

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

Goal

The goal of this tutorial is to learn how to perform logic operations on images.

Objectives

- *Explore the roipoly function to generate image masks.*
- *Learn how to logically AND two images using the bitand function.*
- *Learn how to logically OR two images using the bitor function.*
- *Learn how to obtain the negative of an image using the bitcmp function.*
- *Learn how to logically XOR two images using the bitxor function.*

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

1. Use the MATLAB help system to learn how to use the *roipoly* function when only an image is supplied as a parameter.

Question 1: How do we add points to the polygon?

Question 2: How do we delete points from the polygon?

Question 3: How do we end the process of creating a polygon?

2. Use the *roipoly* function to generate a mask for the pout image.

```
I = imread('pout.tif');
bw = roipoly(I);
```

Question 4: What class is the variable bw?

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

Question 5: What does the variable bw represent?

Logic functions operate at the bit level; that is, the bits of each image pixel are compared individually, and the new bit is calculated based on the operator we are using (AND, OR, or XOR). This means that we can compare only two images that have the same number of bits per pixel as well as equivalent dimensions. In order for us to use the *bw* image in any logical calculation, we must ensure that it consists of the same number of bits as the original image. Because the *bw* image already has the correct number of rows and columns, we need to convert only the image to *uint8*, so that each pixel is represented by 8 bits.

3. Convert the mask image to class uint8.

```
bw2 = uint8(bw);
```

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

Question 6: In the above conversion step, what would happen if we used the im2uint8 function to convert the bw image as opposed to just using uint8(bw)?

(Hint: after conversion, check what is the maximum value of the image bw2.)

4. Use the *bitand* function to compute the logic AND between the original image and the new mask image.

```
I2 = bitand(I,bw2);
imshow(I2);
```

Question 7: What happens when we logically AND the two images?

To see how to OR two images, we must first visit the *bitcmp* function, which is used for complementing image bits (NOT).

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

5. Use the *bitcmp* function to generate a complemented version of the *bw2* mask.

```
bw_cmp = bitcmp(bw2);
figure;
subplot(1,2,1), imshow(bw2), title('Original Mask');
subplot(1,2,2), imshow(bw_cmp), title('Complemented Mask');
```

Question 8: What happened when we complemented the bw2 image?

We can now use the complemented mask in conjunction with *bitor*.

6. Use *bitor* to compute the logic OR between the original image and the complemented mask.

```
I3 = bitor(I,bw_cmp);
figure, imshow(I3);
```

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

Question 9: Why did we need to complement the mask? What would have happened if we used the original mask to perform the OR operation?

The IPT also includes function *imcomplement*, which performs the same operation as the *bitcmp* function, complementing the image. The function *imcomplement* allows input images to be binary, grayscale, or RGB, whereas *bitcmp* requires that the image be an array of unsigned integers.

7. Complement an image using the *imcomplement* function.

```
bw_cmp2 = imcomplement(bw2);
```

Question 10: How can we check to see that the bw_cmp2 image is the same as the bw_cmp image?

The XOR operation is commonly used for finding differences between two images.

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

8. Close all open figures and clear all workspace variables.
9. Use the *bitxor* function to find the difference between two images.

```
I = imread('cameraman.tif');
I2 = imread('cameraman2.tif');
I_xor = bitxor(I,I2);
figure;
subplot(1,3,1), imshow(I), title('Image 1');
subplot(1,3,2), imshow(I2), title('Image 2');
subplot(1,3,3), imshow(I_xor,[]), title('XOR Image');
```

Logic operators are often combined to achieve a particular task. In next steps, we will use all the logic operators discussed previously to darken an image only within a region of interest.

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

10. Close all open figures and clear all workspace variables.
11. Read in image and calculate an adjusted image that is darker using the *imdivide* function.

```
I = imread('lindsay.tif');
I_adj = imdivide(I,1.5);
```

12. Generate a mask by creating a region of interest polygon.

```
bw = im2uint8(roipoly(I));
```

13. Use logic operators to show the darker image only within the region of interest, while displaying the original image elsewhere.

```
bw_cmp = bitcmp(bw); %mask complement
roi = bitor(I_adj,bw_cmp); %roi image
```

5.4 TUTORIAL: LOGIC OPERATIONS AND REGION OF INTEREST PROCESSING

```
not_roi = bitor(I,bw); %non_roi image

new_img = bitand(roi,not_roi); %generate new image

imshow(new_img) %display new image
```

Question 11: How could we modify the above code to display the original image within the region of interest and the darker image elsewhere?

WHAT HAVE WE LEARNED?

- *Arithmetic operations* can be used to blend two images (addition), detect differences between two images or video frames (subtraction), increase an image's average brightness (multiplication/division by a constant), among other things.
- When performing any arithmetic image processing operation, pay special attention to *the data types involved*, *their ranges*, and *the desired way to handle overflow* and *underflow situations*.
- MATLAB's IPT has built-in functions for image addition (*imadd*), subtraction (*imsubtract* and *imabsdiff*), multiplication (*immultiply*), and division (*imdivide*). It also has a function (*imlincomb*) that can be used to perform several arithmetic operations without having to worry about underflow or overflow of intermediate results.

WHAT HAVE WE LEARNED?

- Logic operations are performed on a bit-by-bit basis and are often used to mask out a portion of an image (*the region of interest*) for further processing.
- MATLAB's IPT has built-in functions for performing basic logic operations on digital images: AND (*bitand*), OR (*bitor*), NOT (*bitcmp*), and XOR (*bitxor*).

PROBLEMS

Problem 1. What would be the result of adding a positive constant (scalar) to a monochrome image?

Problem 2. What would be the result of subtracting a positive constant (scalar) from a monochrome image?

Problem 3. What would be the result of multiplying a monochrome image by a positive constant greater than 1.0?

Problem 4. What would be the result of multiplying a monochrome image by a positive constant less than 1.0?

PROBLEMS

Problem 5. What happens when you add a uint8 [0, 255] monochrome image to itself?

Problem 6. What happens when you multiply a uint8 [0, 255] monochrome image by itself?

Problem 7. Would pixel-by-pixel division be a better way to find the differences between two monochrome images than subtraction, absolute difference, or XOR? Explain.

PROBLEMS

Problem 8. Write a MATLAB function to perform brightness correction on monochrome images. It should take as arguments a monochrome image, a number between 0 and 100 (amount of brightness correction, expressed in percentage terms), and a third parameter indicating whether the correction is intended to brighten or darken the image.

5.5 INTRODUCTION

- Geometric operations modify *the geometry of an image by repositioning pixels in a constrained way*.
- In other words, rather than changing the pixel values of an image (as most techniques studied in Part I of this lecture do), they modify the spatial relationships between groups of pixels representing features or objects of interest within the image.

5.5 INTRODUCTION

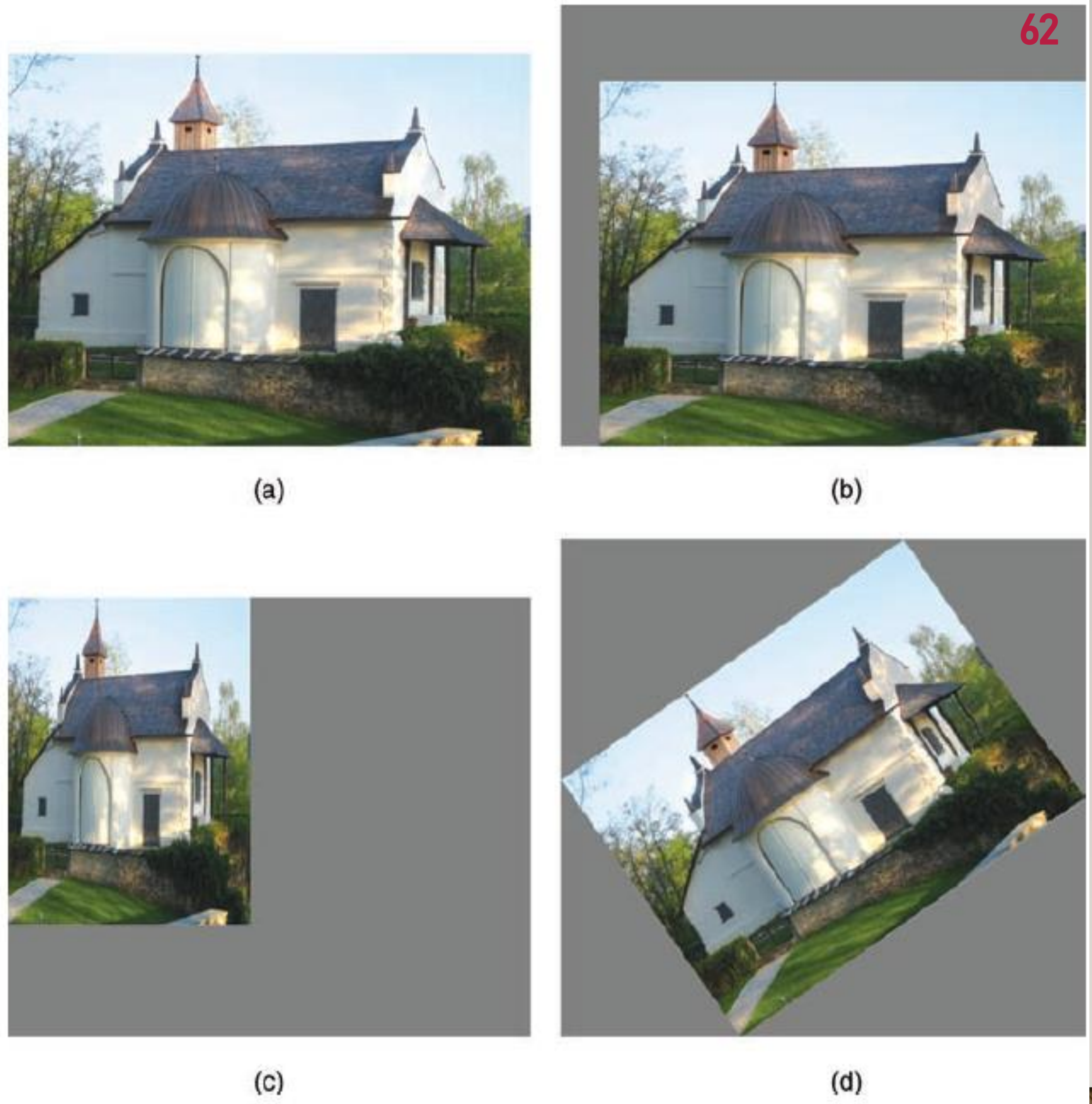


FIGURE 7.1 Examples of typical geometric operations: (a) original image; (b) translation (shifting); (c) scaling (resizing); (d) rotation.

5.5 INTRODUCTION

- Geometric operations can be used to accomplish different goals, such as the following:
  - Correcting geometric distortions introduced during the image acquisition process (e.g., due to the use of a fish-eye lens).
  - Creating special effects on existing images, such as twirling, bulging, or squeezing a picture of someone's face.
  - As part of *image registration*—the process of matching the common features of two or more images of the same scene, acquired from different viewpoints or using different equipment.

5.5 INTRODUCTION

- Most geometric operations consist of two basic components:
  1. Mapping Function: This is typically specified using a set of spatial transformation equations (and a procedure to solve them).
  2. Interpolation Methods: These are used to compute the new value of each pixel in the spatially transformed image.

5.6 MAPPING AND AFFINE TRANSFORMATIONS

- A geometric operation can be described mathematically as the process of transforming an input image $f(x, y)$ into a new image $g(x', y')$ by modifying *the coordinates of image pixels*:

$$f(x, y) \rightarrow g(x', y')$$

that is, the pixel value originally located at coordinates (x, y) will be relocated to coordinates (x', y') in the output image.

- To model this process, *a mapping function* is needed. The mapping function specifies the new coordinates (in the output image) for each pixel in the input image:

$$(x', y') = T(x, y)$$

5.6 MAPPING AND AFFINE TRANSFORMATIONS

- This mapping function is an arbitrary 2D function. It is often specified as two separate functions, one for each dimension:

$$x' = T_x(x, y)$$

and

$$y' = T_y(x, y)$$

where T_x and T_y are usually expressed as polynomials in x and y. The case where T_x and T_y are linear combinations of x and y is called *affine transformation* (or *affine mapping*):

$$x' = a_0x + a_1y + a_2$$
$$y' = b_0x + b_1y + b_2$$

5.6 MAPPING AND AFFINE TRANSFORMATIONS

- A general equation can also be expressed in matrix form as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ b_0 & b_1 & b_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

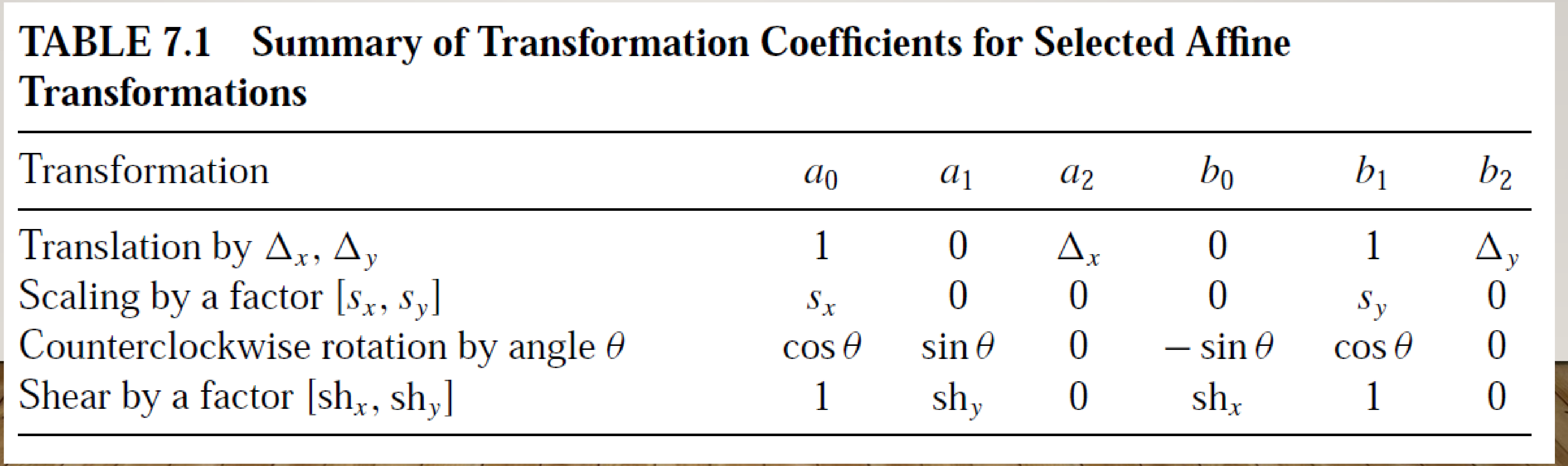


TABLE 7.1 Summary of Transformation Coefficients for Selected Affine Transformations

Transformation	a_0	a_1	a_2	b_0	b_1	b_2
Translation by Δ_x, Δ_y	1	0	Δ_x	0	1	Δ_y
Scaling by a factor $[s_x, s_y]$	s_x	0	0	0	s_y	0
Counterclockwise rotation by angle θ	$\cos\theta$	$\sin\theta$	0	$-\sin\theta$	$\cos\theta$	0
Shear by a factor $[\text{sh}_x, \text{sh}_y]$	1	sh_y	0	sh_x	1	0

5.6 MAPPING AND AFFINE TRANSFORMATIONS

In MATLAB

- The IPT has two functions associated with affine transforms: *maketform* and *imtransform*.
- The *maketform* function is used to define the desired 2D spatial transformation. It creates a MATLAB structure (called a *TFORM*) that contains all the parameters required to perform the transformation. In addition to affine transformations, *maketform* also supports the creation of projective and custom transformations.
- After having defined the desired transformation, it can be applied to an input image using function *imtransform*.

EXAMPLE 5.4

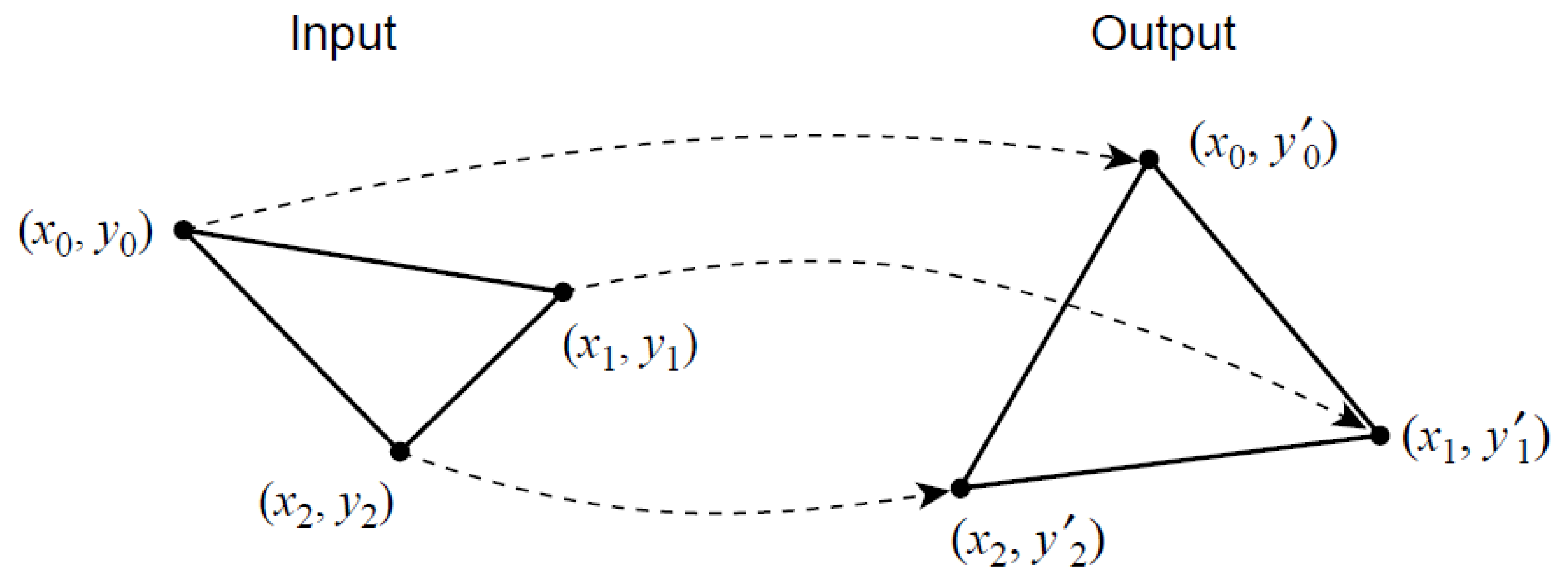


FIGURE 7.2 Mapping one triangle onto another by an affine transformation.

EXAMPLE 5.4

Generate the affine transformation matrix for each of the operations below: *(a) rotation by 30º;*

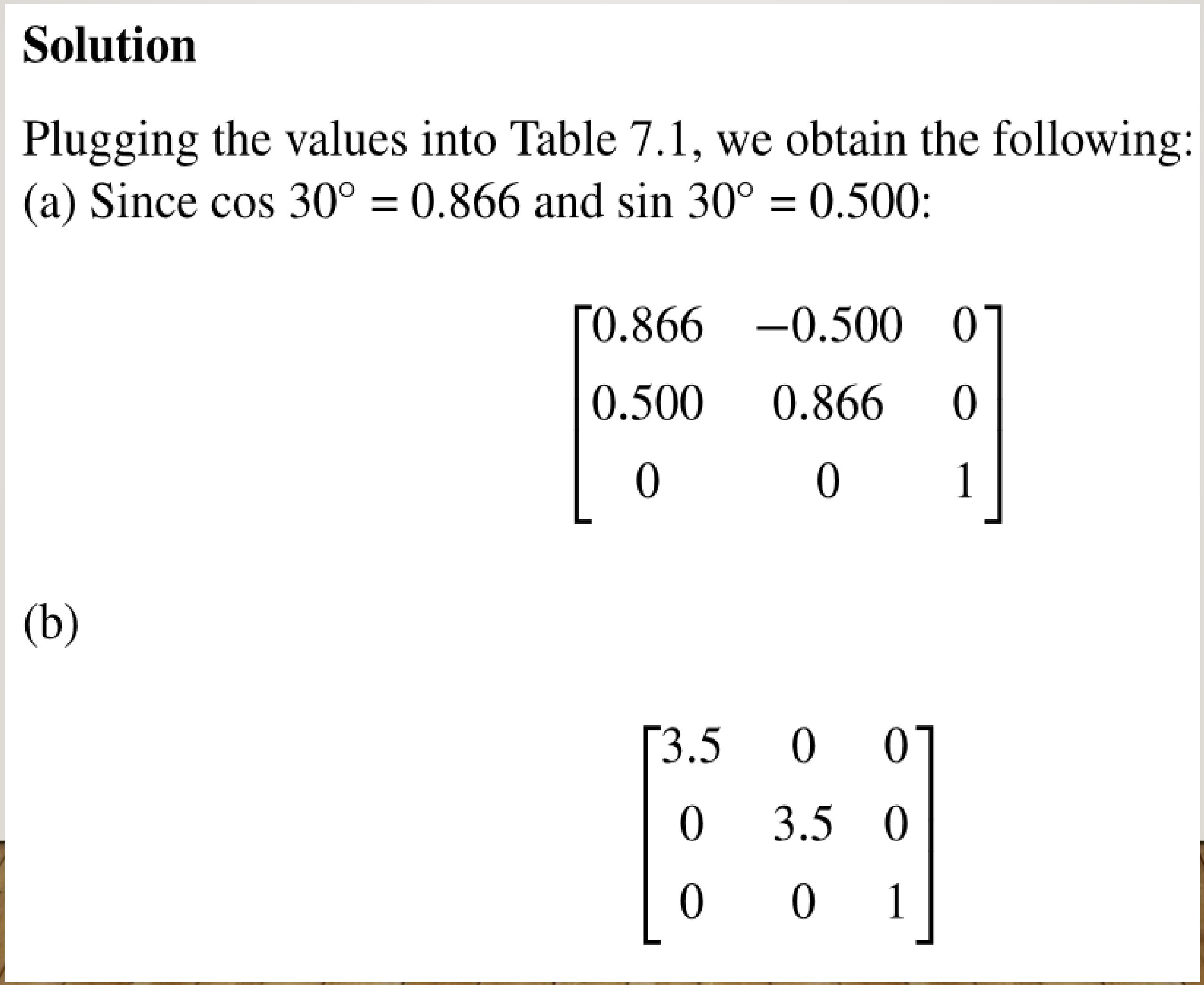
(b) scaling by a factor 3.5 in both dimensions;

(c) translation by [25, 15] pixels;

(d) shear by a factor [2, 3].

Use MATLAB to apply the resulting matrices to an input image of your choice.

EXAMPLE 5.4



Solution

Plugging the values into Table 7.1, we obtain the following:
(a) Since cos 30° = 0.866 and sin 30° = 0.500:

$$\begin{bmatrix} 0.866 & -0.500 & 0 \\ 0.500 & 0.866 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 3.5 & 0 & 0 \\ 0 & 3.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

EXAMPLE 5.4

(c)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 25 & 15 & 1 \end{bmatrix}$$

(d)

$$\begin{bmatrix} 1 & 3 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

EXAMPLE 5.4

MATLAB code:

```
        filename = 'any image of your choice'
        I = imread(filename);
% Rotation
        Ta = maketform('affine', ...
        [cosd(30)   -sind(30)   0; sind(30)   cosd(30)   0; 0   0   1]');
        Ia = imtransform(I,Ta);
%Scaling
        Tb = maketform('affine',[3.5   0   0; 0   3.5   0; 0   0   1]');
        Ib = imtransform(I,Tb);
```

EXAMPLE 5.4

```
% Translation
        xform = [1   0   25; 0   1   15; 0   0   1]';
        Tc = maketform('affine', xform);
        Ic = imtransform(I, Tc, 'XData', ...
        [1 (size(I,2)+xform(3,1))], 'YData', ...
        [1 (size(I,1)+xform(3,2))], 'FillValues', 128 );
% Shearing
        Td = maketform('affine',[1   3   0; 2   1   0; 0   0   1]');
        Id = imtransform(I,Td);
```

5.7 INTERPOLATION METHODS

The Need for Interpolation

- After a geometric operation has been performed on the original image, the resulting value for each pixel can be computed in two different ways. The first one is called *forward mapping*—also known as source-to-target mapping (Figure 7.3)—and consists of iterating over every pixel of the input image, computing its new coordinates, and copying the value to the new location.

5.7 INTERPOLATION METHODS

The Need for Interpolation

- This approach has a number of problems, such as the following:
  - *Many coordinates calculated by the transformation equations are not integers, and need to be rounded off to the closest integer to properly index a pixel in the output image.*
  - *Many coordinates may lie out of bounds (e.g., negative values).*
  - *Many output pixels' coordinates are addressed several times during the calculations (which is wasteful) and some are not addressed at all (which leads to "holes" in the output image, meaning that no pixel value was computed for that coordinate pair).*

5.7 INTERPOLATION METHODS

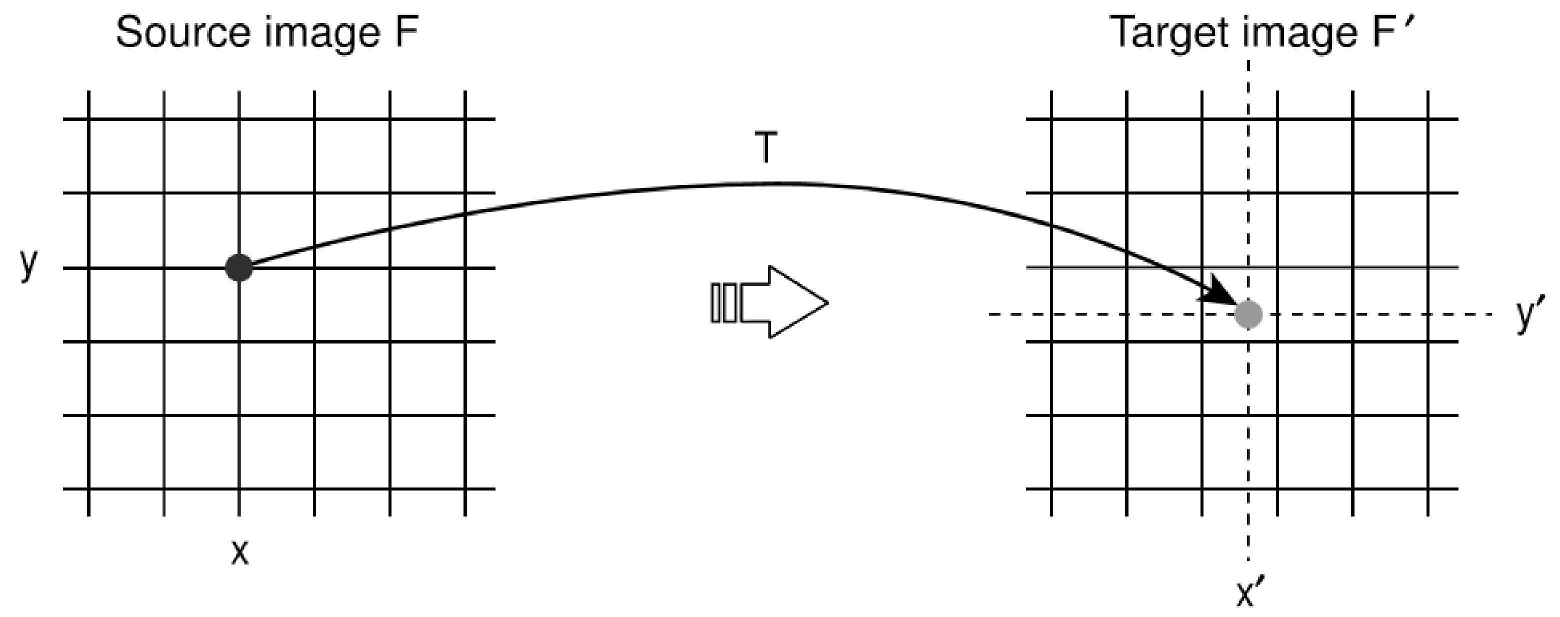


FIGURE 7.3 Forward mapping: for each pixel position in the input image, the corresponding (continuous) target position—resulting from applying a geometric transformation T—is found in the output image. In general, the target position (x', y') does not coincide with any discrete raster point, and the value of the pixel in the input image is copied to one of the adjacent target pixels. Redrawn from [BB08].

5.7 INTERPOLATION METHODS

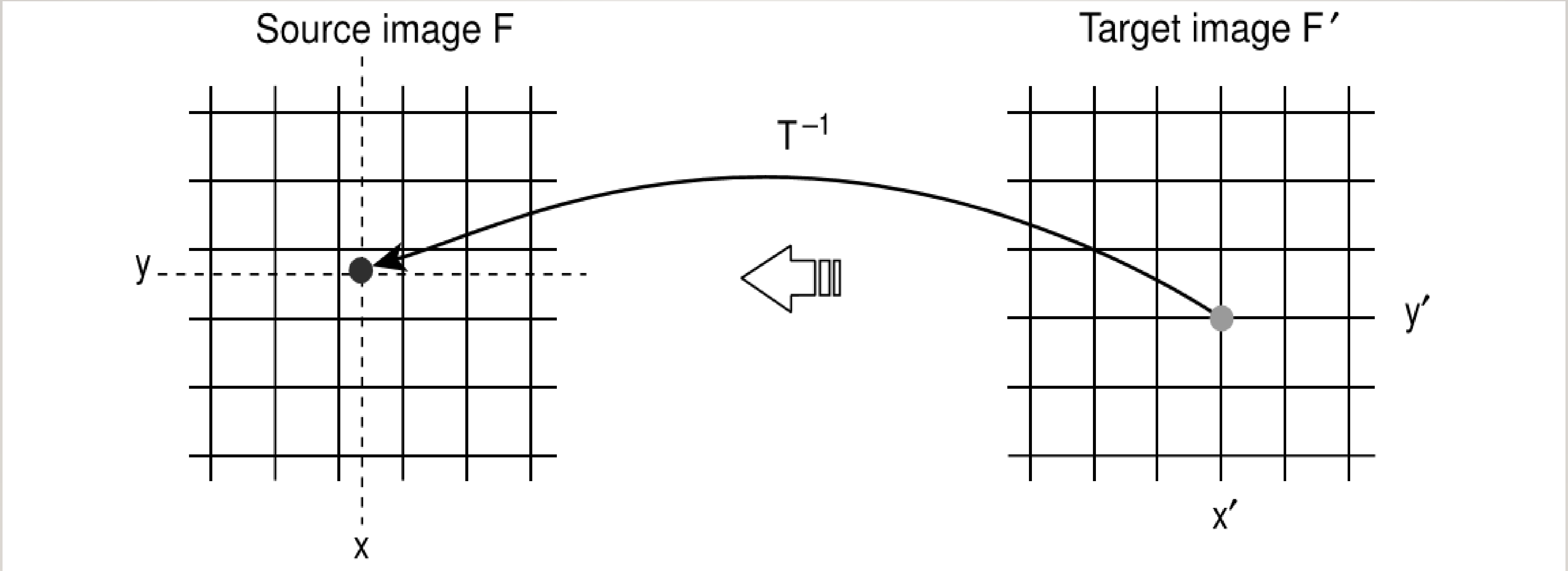


FIGURE 7.4 Backward mapping: for each discrete pixel position in the output image, the corresponding continuous position in the input image (x, y) is found by applying the inverse mapping function T^{-1}. The new pixel value is found by interpolation among the neighbors of (x, y) in the input image. Redrawn from [BB08].

5.7 INTERPOLATION METHODS

- The solution to the limitations listed above usually comes in the form of *a backward mapping*—also known as *target-to-source mapping* (Figure 7.4)—approach, which consists of visiting every pixel in the output image and applying the inverse transformation to determine the coordinates in the input image from which a pixel value must be sampled.
- Since this backward mapping process often results in coordinates outside the sampling grid in the original image, it usually requires some type of *interpolation* to compute the best value for that pixel.

5.7.1 A SIMPLE APPROACH TO INTERPOLATION

- If you were asked to write code to enlarge or reduce an image by a certain factor (e.g., a factor of 2 in both directions), you would probably deal with the problem of removing pixels (in the case of shrinking) by *subsampling the original image* by a factor of 2 in both dimensions, that is, skipping every other pixel along each row and column.

- Conversely, for the task of enlarging the image by a factor of 2 in both dimensions, you would probably *opt* for copying each original pixel to an n x n block in the output image.

5.7.1 A SIMPLE APPROACH TO INTERPOLATION

- These simple interpolation schemes (pixel removal and pixel duplication, respectively) are fast and easy to understand, but suffer from several limitations, such as the following:
  - ➢ *The "blockiness" effect that may become noticeable when enlarging an image.*
  - ➢ *The possibility of removing essential information in the process of shrinking an image.*
  - ➢ *The difficulty in extending these approaches to arbitrary, non-integer, resizing factors.*
- Other simplistic methods—such as using the mean (or median) value of *the original n x n block* in the input image to determine the value of each output pixel in the shrunk image—also produce low-quality results and are bound to fail in some cases. These limitations call for improved interpolation methods, which will be briefly described next.

5.7.2 ZERO-ORDER (NEAREST-NEIGHBOR) INTERPOLATION

- This baseline interpolation scheme rounds off the calculated coordinates (x', y') to their nearest integers.

- Zero-order (or nearest-neighbor) interpolation is simple and computationally fast, but produces low-quality results, with artifacts such as *blockiness effects*—which are more pronounced at large-scale factors—and jagged straight lines—particularly after rotations by angles that are not multiples of 90° (see Figure 7.5b).

5.7.3 FIRST-ORDER (BILINEAR) INTERPOLATION

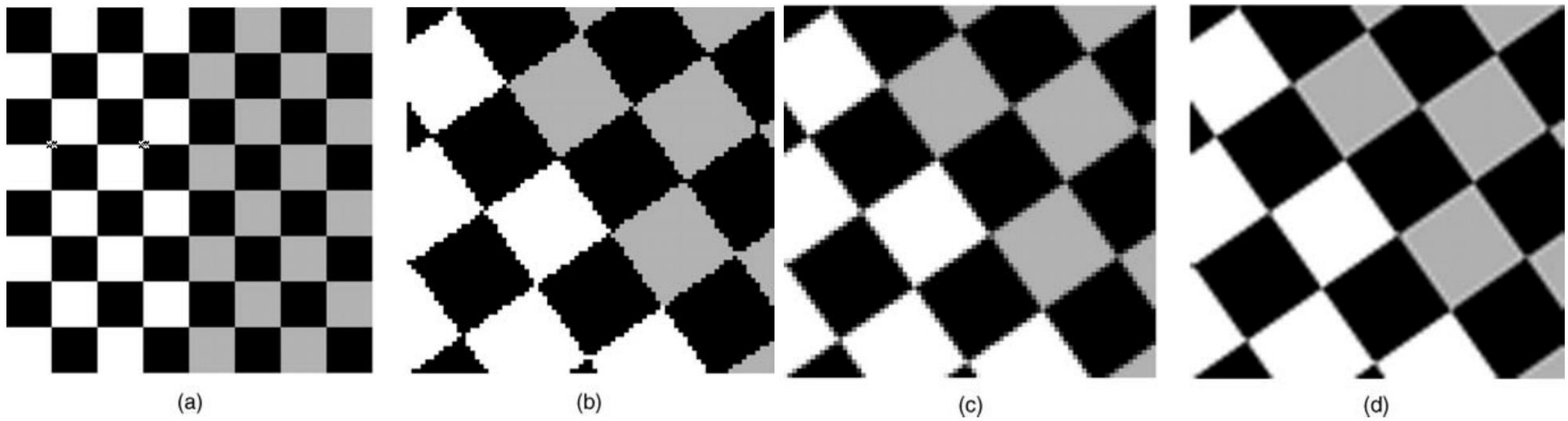
- First-order (or bilinear) interpolation calculates the gray value of the interpolated pixel (at coordinates (x', y')) as a weighted function of the gray values of the four pixels surrounding the reference pixel in the input image.
- Bilinear interpolation produces visually better results than the nearest-neighbor interpolation at the expense of additional CPU time (see Figure 7.5c).

5.7.4 HIGHER ORDER INTERPOLATIONS

- Higher order interpolations are more sophisticated—and computationally expensive—methods for *interpolating the gray value of a pixel*. The third-order interpolation scheme implemented in several MATLAB functions is also known as bicubic interpolation. It takes into account the 4x4 neighborhood around the reference pixel and computes the resulting gray level of the interpolated pixel by performing the convolution of the 4x4 neighborhood with a cubic function.
- Figure 7.5 shows the results of using different interpolation schemes to rotate an image by 35º. The jagged edge effect of the zero-order interpolation is visible (in part b), but there is little—if any—perceived difference between the bipolar (part c) and bicubic (part d) results.

5.7.4 HIGHER ORDER INTERPOLATIONS

***FIGURE 7.5** Effects of different interpolation techniques on rotated images: (a) original image; zoomed-in versions of rotated (35°) image using (b) zero-order (nearest-neighbor) interpolation; (c) first-order (bilinear) interpolation; (d) third-order (bicubic) interpolation.*



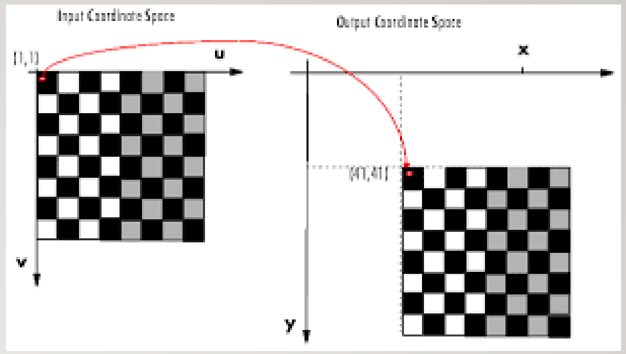
5.8 GEOMETRIC OPERATIONS USING MATLAB

- In this section, we will present a summary of typical geometric operations involving digital images that can easily be implemented using MATLAB and the IPT.

5.8.1 Zooming, Shrinking, and Resizing

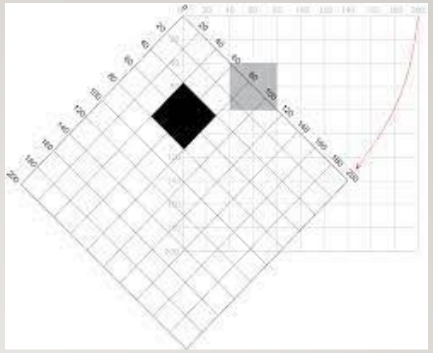
- One of the most common geometric operations is the resize operation. In this lecture, we distinguish between *true image resizing*—where the resulting image size (in pixels) is changed—and *resizing an image for human viewing*—which we will refer to as *zooming (in)* and *shrinking* (or *zooming out*). They are both useful image processing operations and often rely on the same underlying algorithms. The main difference lies in the fact that zooming and shrinking are usually performed interactively (with a tool such as IPT's *imshow* or *imtool*) and their results last for a brief moment, whereas resizing is typically accomplished in a noninteractive way (e.g., as part of a MATLAB script) and its results are stored for longer term use.

5.8.1 TRANSLATION



- Translation of an input image $f(x, y)$ with respect to its Cartesian origin to produce an output image $g(x', y')$ where each pixel is displaced by $[\Delta x, \Delta y]$ (i.e., $x' = x + \Delta x$ and $y' = y + \Delta y$) consists of a special case of affine transform.

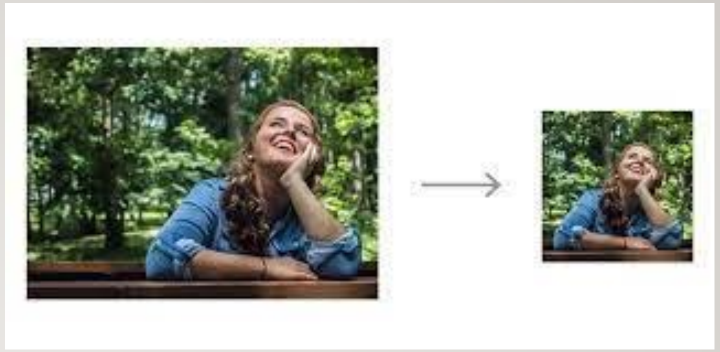
5.8.2 ROTATION



- Rotation of an image constitutes another special case of affine transform (as discussed in Section 5.2). Consequently, image rotation can also be accomplished using *maketform* and *imtransform*.

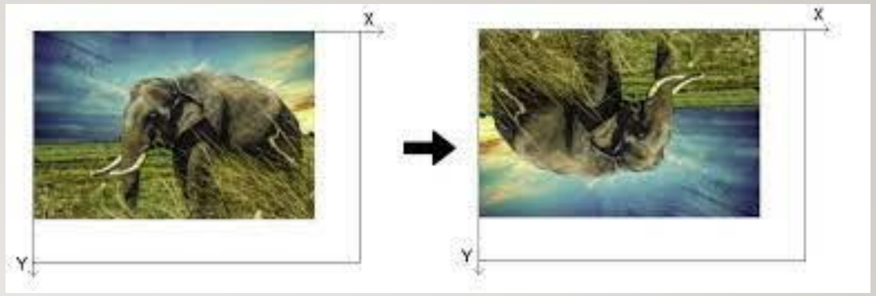
- The IPT also has a specialized function for rotating images, *imrotate*. Similar to *imresize*, *imrotate* allows the user to specify the interpolation method used: nearest-neighbor (the default method), bilinear, or bicubic. It also allows specification of the size of the output image.

5.8.3 CROPPING



- The IPT has a function for cropping images, *imcrop*, which crops an image to a specified rectangle. The crop rectangle can be specified interactively (with the mouse) or its coordinates be passed as parameters to the function.

5.8.4 FLIPPING



- The IPT has two functions for flipping matrices (which can also be used for raster images, of course): *flipud*—which flips a matrix up to down—and *fliplr*—which flips a matrix left to right.

5.9 OTHER GEOMETRIC OPERATIONS AND APPLICATIONS

5.9.1 Warping

5.9.2 Nonlinear Image Transformations (Twirling, Rippling)

5.9.3 Morphing

5.9.4 Seam Carving

5.9.5 Image Registration

5.9 OTHER GEOMETRIC OPERATIONS AND APPLICATIONS

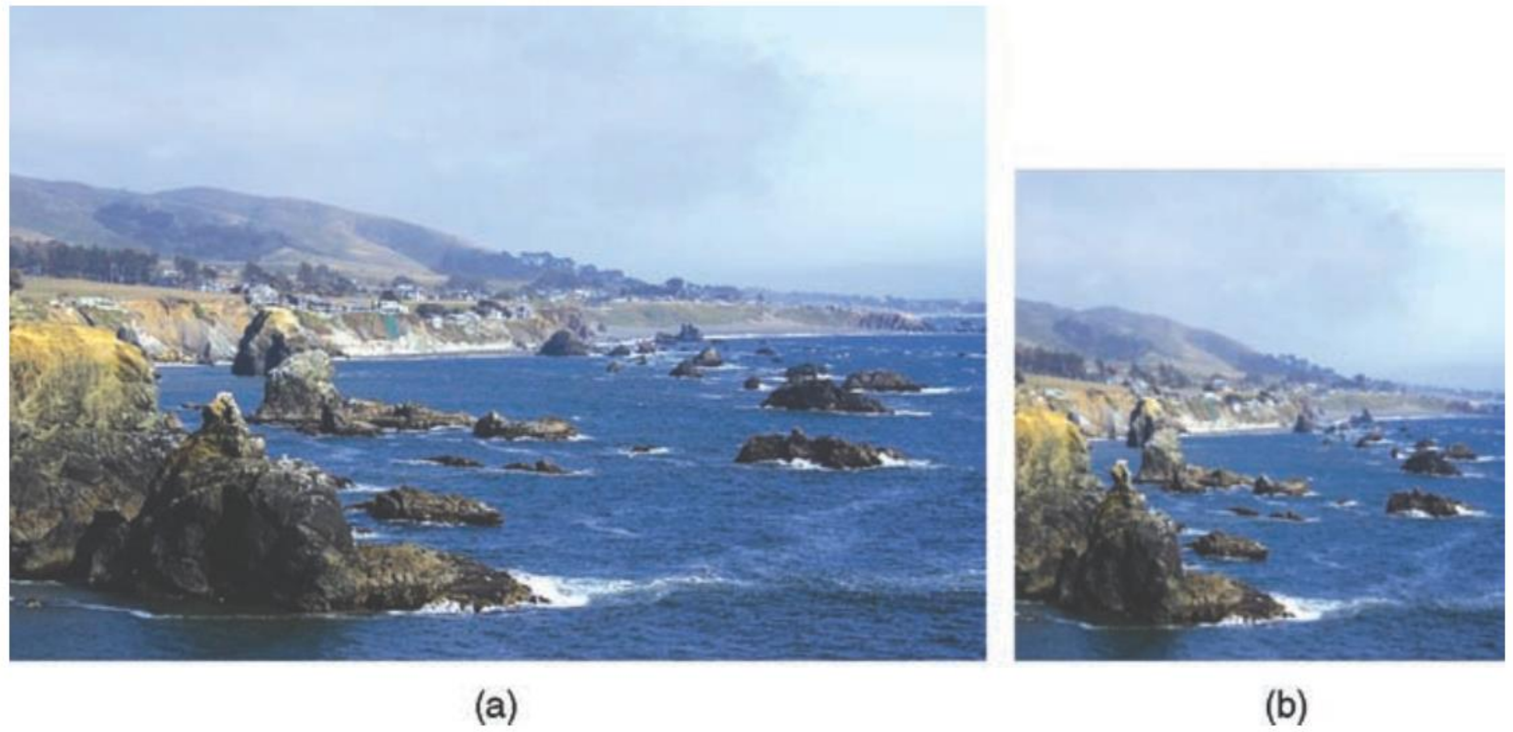


FIGURE 7.7 Using seam carving for content-aware resizing: (a) original image (334 × 500 pixels); (b) cropped image (256 × 256 pixels). Original image from Flickr. Seam carving results were obtained using the publicly available implementation by Mathias Lux: http://code.google.com/p/java-imageseams/.

5.9 OTHER GEOMETRIC OPERATIONS AND APPLICATIONS

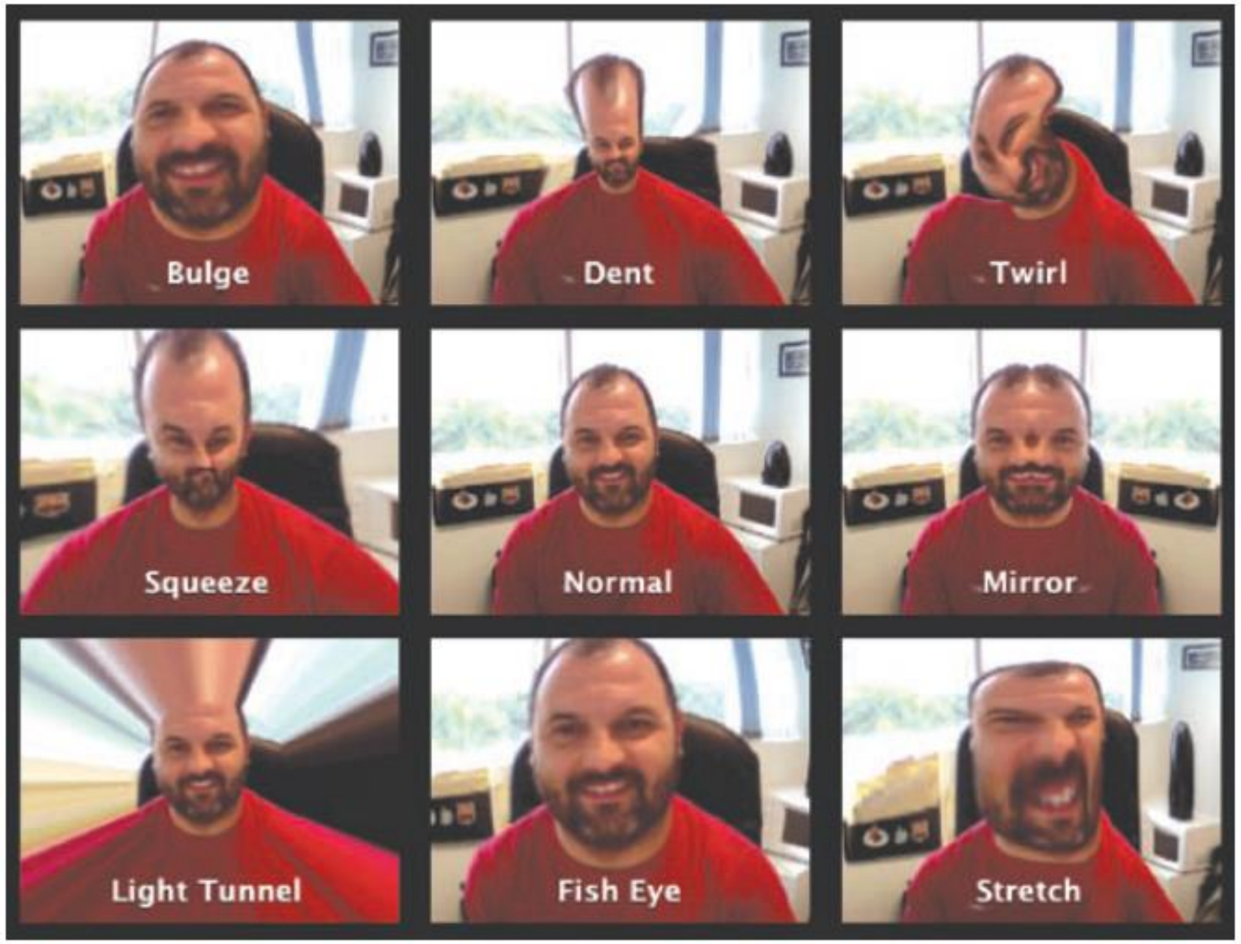


FIGURE 7.6 Image deformation effects using *Photo Booth*.

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

Goal

The goal of this tutorial is to learn how to crop, resize, and rotate digital images.

Objectives

- *Learn how to crop an image using the imcrop function.*
- *Learn how to resize an image using the imresize function.*
- *Learn how to flip an image upside down and left–right using flipud and fliplr.*
- *Learn how to rotate an image using the imrotate function.*
- *Explore interpolation methods for resizing and rotating images.*

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

1. Open the *nature* image and use the *Crop Image* option in the *Image Tool* (*imtool*) toolbar to crop it such that only the portion of the image containing the house in the background is selected to become the cropped image.

Pay attention to (and write down) the coordinates of the top left and bottom right corners as you select the rectangular area to be cropped. You will need this information for the next step.

2. Double-click inside the selected area to complete the cropping operation.
3. Save the resulting image using the File > Save as... option in the *imtool* menu. Call it *cropped_nature.jpg*.

```
I = imread(nature.jpg');
imtool(I);
```

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

Question 1: Which numbers did you record for the top left and bottom right coordinates and what do they mean?

Hint: Pay attention to the convention used by the Pixel info status bar at the bottom of the *imtool* main window. The IPT occasionally uses a so-called spatial coordinate system, whereas *y* represents rows and *x* represents columns. This does not correspond to the image axis coordinate system defined in Chapter 2.

4. Open and display the cropped image.

```
I2 = imread('cropped_nature.jpg');
imshow(I2);
```

5. We shall now use the coordinates recorded earlier to perform a similar cropping from a script.

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

6. The *imcrop* function expects the crop rectangle—a four-element vector [*xmin ymin width height*]—to be passed as a parameter.

7. Perform the steps below replacing my values for x1, y1, x2, and y2 with the values you recorded earlier.

```
x1 = 186; x2 = 211; y1 = 105; y2 = 159;
xmin = x1; ymin = y1; width = x2-x1; height = y2-y1;
I3 = imcrop(I, [xmin ymin width height]);
imshow(I3);
```

Resizing an image consists of enlarging or shrinking it, using *nearest-neighbor*, *bilinear*, or *bicubic* interpolation. Both resizing procedures can be executed using the *imresize* function. Let us first explore enlarging an image.

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

8. Enlarge the nature image by a scale factor of 3. By default, the function uses bicubic interpolation.

```
I_big1 = imresize(I,3);
figure, imshow(I), title('Original Image');
figure, imshow(I_big1), ...
title('Enlarged Image w/ bicubic interpolation');
```

As you have seen in Chapter 4, the IPT function *imtool* can be used to inspect the pixel values of an image. The *imtool* function provides added functionality to visual inspection of images, such as zooming and pixel inspection.

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

9. Use the *imtool* function to inspect the resized image, I_big1.

```
imtool(I_big1)
```

10. Scale the image again using nearest-neighbor and bilinear interpolations.

```
I_big2 = imresize(I,3,'nearest');
I_big3 = imresize(I,3,'bilinear');
figure, imshow(I_big2), ...
title('Resized w/ nearest-neighbor interpolation');
figure, imshow(I_big3), ...
title('Resized w/ bilinear interpolation');
```

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

Question 2: Visually compare the three resized images. How do they differ?

One way to shrink an image is by simply deleting rows and columns of the image.

11. Close any open figures.

12. Reduce the size of the cameraman image by a factor of 0.5 in both dimensions.

```
I_rows = size(I,1);
I_cols = size(I,2);
I_sm1 = I(1:2:I_rows, 1:2:I_cols);
figure, imshow(I_sm1);
```

Question 3: How did we scale the image?

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

Question 4: What are the limitations of this technique?

Although the technique above is computationally efficient, its limitations may require us to use another method. Just as we used the *imresize* function for enlarging, we can just as well use it for shrinking.

When using the *imresize* function, a scale factor larger than 1 will produce an image larger than the original, and a scale factor smaller than 1 will result in an image smaller than the original.

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

13. Shrink the image using the *imresize* function.

```
I_sm2 = imresize(I,0.5,'nearest');
I_sm3 = imresize(I,0.5,'bilinear');
I_sm4 = imresize(I,0.5,'bicubic');
figure, subplot(1,3,1), imshow(I_sm2), ...
title('Nearest-neighbor Interpolation');
subplot(1,3,2), imshow(I_sm3), title('Bilinear Interpolation');
subplot(1,3,3), imshow(I_sm4), title('Bicubic Interpolation');
```

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

Note that in the case of shrinking using either bilinear or bicubic interpolation, the *imresize* function automatically applies a low-pass filter to the image (whose default size is 11x11), slightly blurring it before the image is interpolated. This helps to reduce the effects of aliasing during resampling (see Chapter 5).

Flipping an image upside down or left–right can be easily accomplished using the *flipud* and *fliplr* functions.

14. Close all open figures and clear all workspace variables.

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

15. Flip the *cameraman* image upside down.

16. Flip the *cameraman* image from left to right.

```
I = imread('nature.jpg');
J = flipud(I);
K = fliplr(I);
subplot(1,3,1), imshow(I), title('Original image');
subplot(1,3,2), imshow(J), title('Flipped upside-down');
subplot(1,3,3), imshow(K), title('Flipped left-right')
```

Rotating an image is achieved through the *imrotate* function.

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

17. Close all open figures and clear all workspace variables.

18. Rotate the eight image by an angle of 35°.

```
I = imread('eight.tif');
I_rot = imrotate(I,35);
imshow(I_rot);
```

Question 5: Inspect the size (number of rows and columns) of I_rot and compare it with the size of I. Why are they different?

Question 6: The previous step rotated the image counterclockwise. How would you rotate the image 35° clockwise?

We can also use different interpolation methods when rotating the image.

5.10 TUTORIAL: IMAGE CROPPING, RESIZING, FLIPPING, AND ROTATION

19. Rotate the same image using bilinear interpolation.

```
I_rot2 = imrotate(I,35,'bilinear');
figure, imshow(I_rot2);
```

Question 7: How did bilinear interpolation affect the output of the rotation?

Hint: The difference is noticeable between the two images near the edges of the rotated image and around the coins.

20. Rotate the same image, but this time crop the output.

```
I_rot3 = imrotate(I,35,'bilinear','crop');
figure, imshow(I_rot3);
```

Question 8: How did the crop setting change the size of our output?

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

1. Open the *nature* image.
2. Use *maketform* to make an affine transformation that resizes the image by a factor [s_x, s_y]. The *maketform* function can accept transformation matrices of various sizes for N-dimensional transformations. But since *imtransform* only performs 2D transformations, you can only specify 3x3 transformation matrices. For affine transformations, the first two columns of the 3 x 3 matrices will have the values a_0, a_1, a_2, b_0, b_1, b_2 from Table 7.1, whereas the last column must contain 0 0 1.
3. Use *imtransform* to apply the affine transformation to the image.
4. Compare the resulting image with the one you had obtained using *imresize*.

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

```
I1 = imread('nature.jpg');
sx = 2; sy = 2;
T = maketform('affine',[sx 0 0; 0 sy 0; 0 0 1]');
I2 = imtransform(I1,T);
imshow(I2), title('Using affine transformation')
I3 = imresize(I1, 2);
figure, imshow(I3), title('Using image resizing');
```

Question 1: Compare the two resulting images (I2 and I3). Inspect size, graylevel range, and visual quality. How are they different? Why?

5. Use *maketform* to make an affine transformation that rotates an image by an angle θ.

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

6. Use *imtransform* to apply the affine transformation to the image.

7. Compare the resulting image with the one you had obtained using *imrotate*.

```
I1 = imread(nature.jpg');
theta = 35*pi/180;
xform = [cos(theta) sin(theta) 0; -sin(theta) cos(theta) 0; 0 0 1]';
T = maketform('affine',xform);
I4 = imtransform(I1, T);
imshow(I4), title('Using affine transformation');
I5 = imrotate(I1, 35);
figure, imshow(I5), title('Using image rotating');
```

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

Question 2: Compare the two resulting images (I4 and I5). Inspect size, gray level range, and visual quality. How are they different? Why?

8. Use *maketform* to make an affine transformation that translates an image by Δx, Δy.

9. Use *imtransform* to apply the affine transformation to the image and use a fill color (average gray in this case) to explicitly indicate the translation.

10. Display the resulting image.

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

```
I1 = imread('nature.jpg');
delta_x = 50;
delta_y = 100;
xform = [1 0 delta_x; 0 1 delta_y; 0 0 1]';
tform_translate = maketform('affine',xform);
I6 = imtransform(I1, tform_translate,...
'XData', [1 (size(I1,2)+xform(3,1))],...
'YData', [1 (size(I1,1)+xform(3,2))],...
'FillValues', 128 );
figure, imshow(I6);
```

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

Question 3: Compare the two images (I1 and I6). Inspect size, gray-level range, and visual quality. How are they different? Why?

11. Use *maketform* to make an affine transformation that performs shearing by a factor [sh_x, sh_y] on an input image.
12. Use *imtransform* to apply the affine transformation to the image.
13. Display the resulting image.

```
I = imread('cameraman.tif');
sh_x = 2; sh_y = 1.5;
xform = [1 sh_y 0; sh_x 1 0; 0 0 1]';
T = maketform('affine',xform);
I7 = imtransform(I1, T);
imshow(I7);
```

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

Image Registration

In the last part of the tutorial, you will learn how to use spatial transformations in the context of image registration.

14. Open the base image (Figure 7.9a) and the unregistered image (Figure 7.9b).

```
base = imread('klcc_a.png');

unregistered = imread('klcc_b.png');
```

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

15. Specify control points in both images using *cpselect* (Figure 7.10). This is an interactive process that is explained in detail in the IPT online documentation.

For the purpose of this tutorial, we will perform the following:

- *Open the Control Point Selection tool.*
- *Choose a zoom value that is appropriate and lock the ratio.*
- *Select the Control Point Selection tool in the toolbar.*
- *Select a total of 10 control points per image, making sure that after we select a point in one image with click on the corresponding point in the other image, thereby establishing a match for that point. See Figure 7.11 for the points I chose.*
- *Save the resulting control points using the File > Export Points to Workspace option in the menu.*

```
cpselect(unregistered, base);
```

5.11 TUTORIAL: SPATIAL TR AND IMAGE REGISTRATION

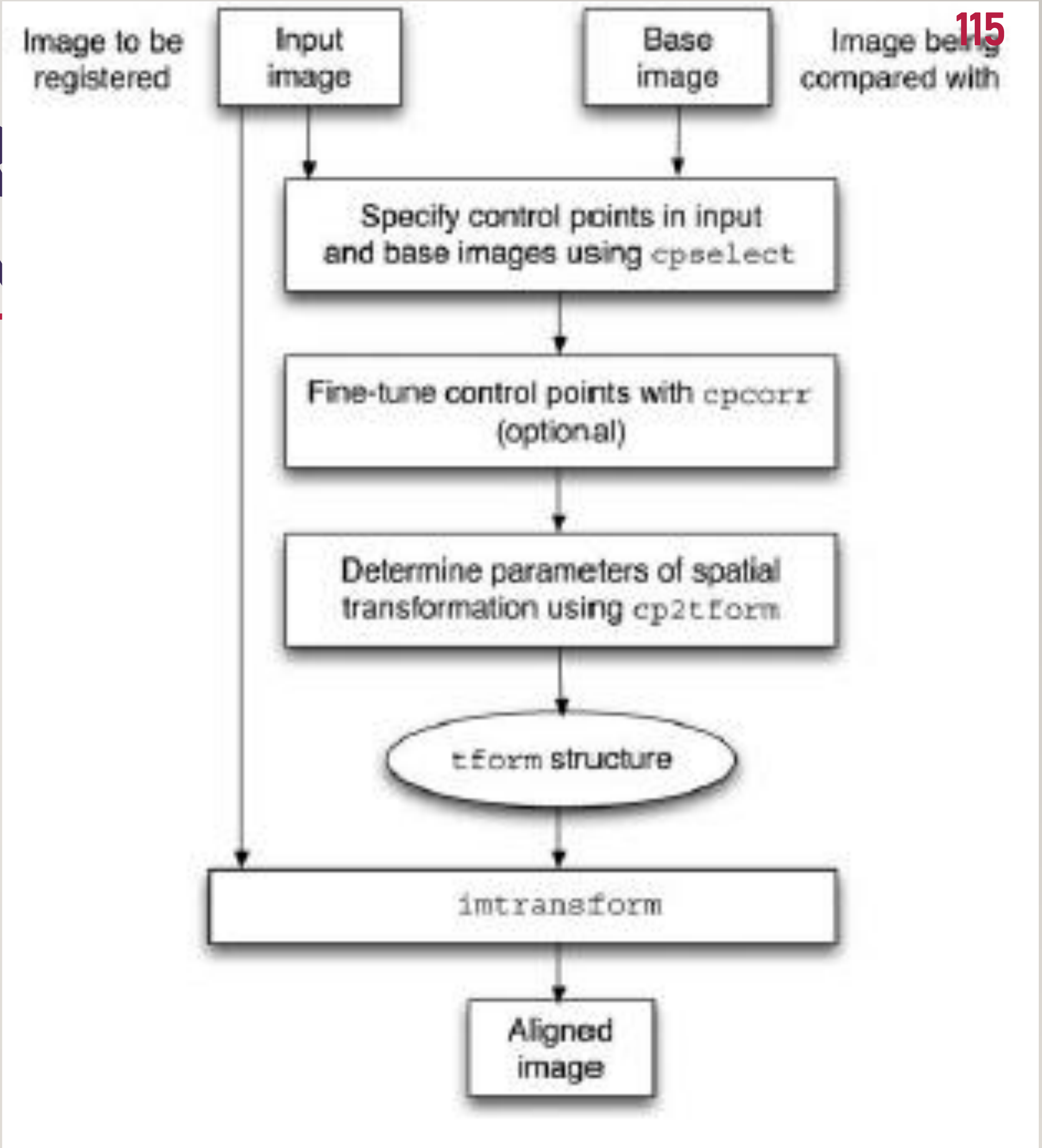


Image to be registered
Input image
Base image
Image being compared with
Specify control points in input and base images using cpselect
Fine-tune control points with cpcorr (optional)
Determine parameters of spatial transformation using cp2tform
tform structure
imtransform
Aligned image

FIGURE 7.8 Image registration using MATLAB and the IPT.

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

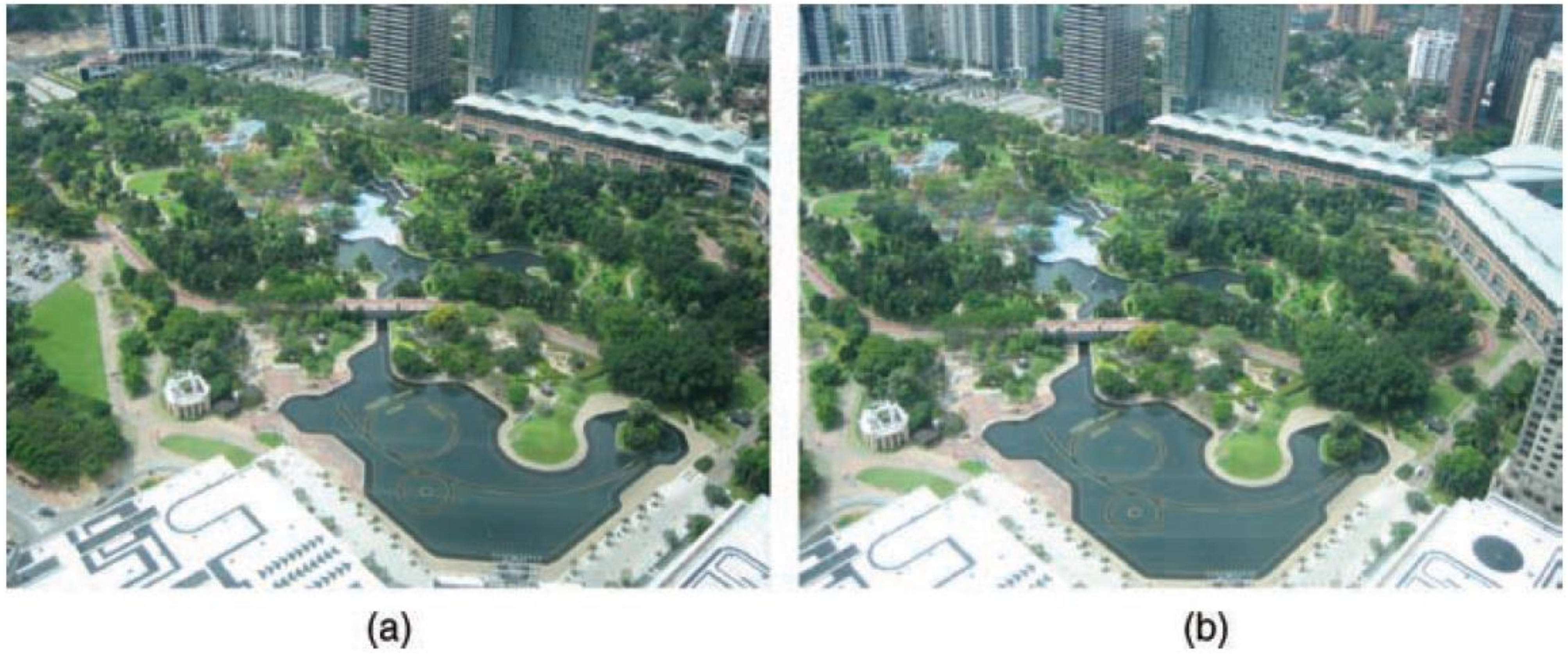


FIGURE 7.9 Interactive image registration: (a) base image; (b) unregistered image.

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

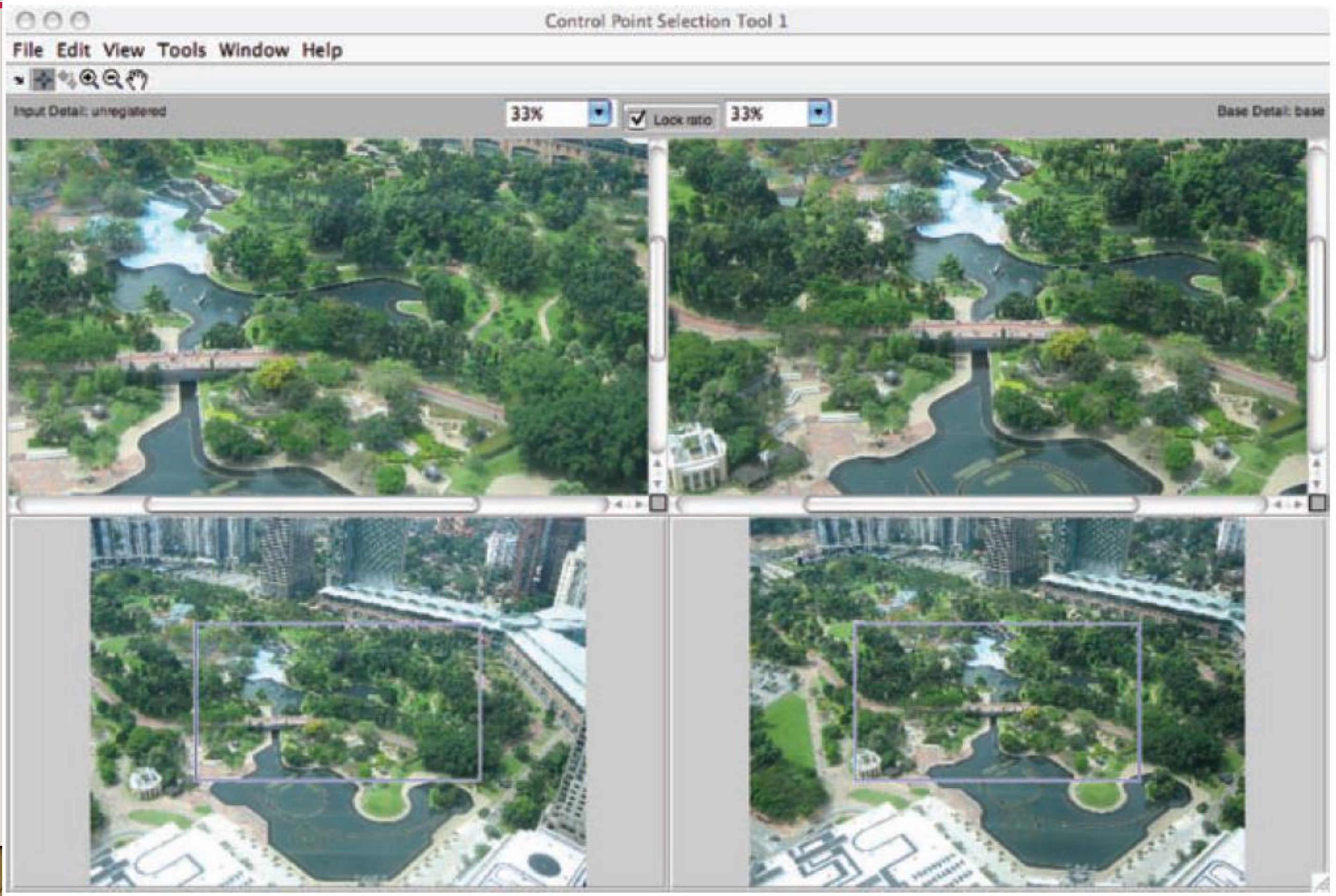

Control Point Selection Tool 1
File Edit View Tools Window Help
Input Detail: unregistered
33%
Lock ratio
33%
Base Detail: base

FIGURE 7.10 The *Control Point Selection* tool.

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

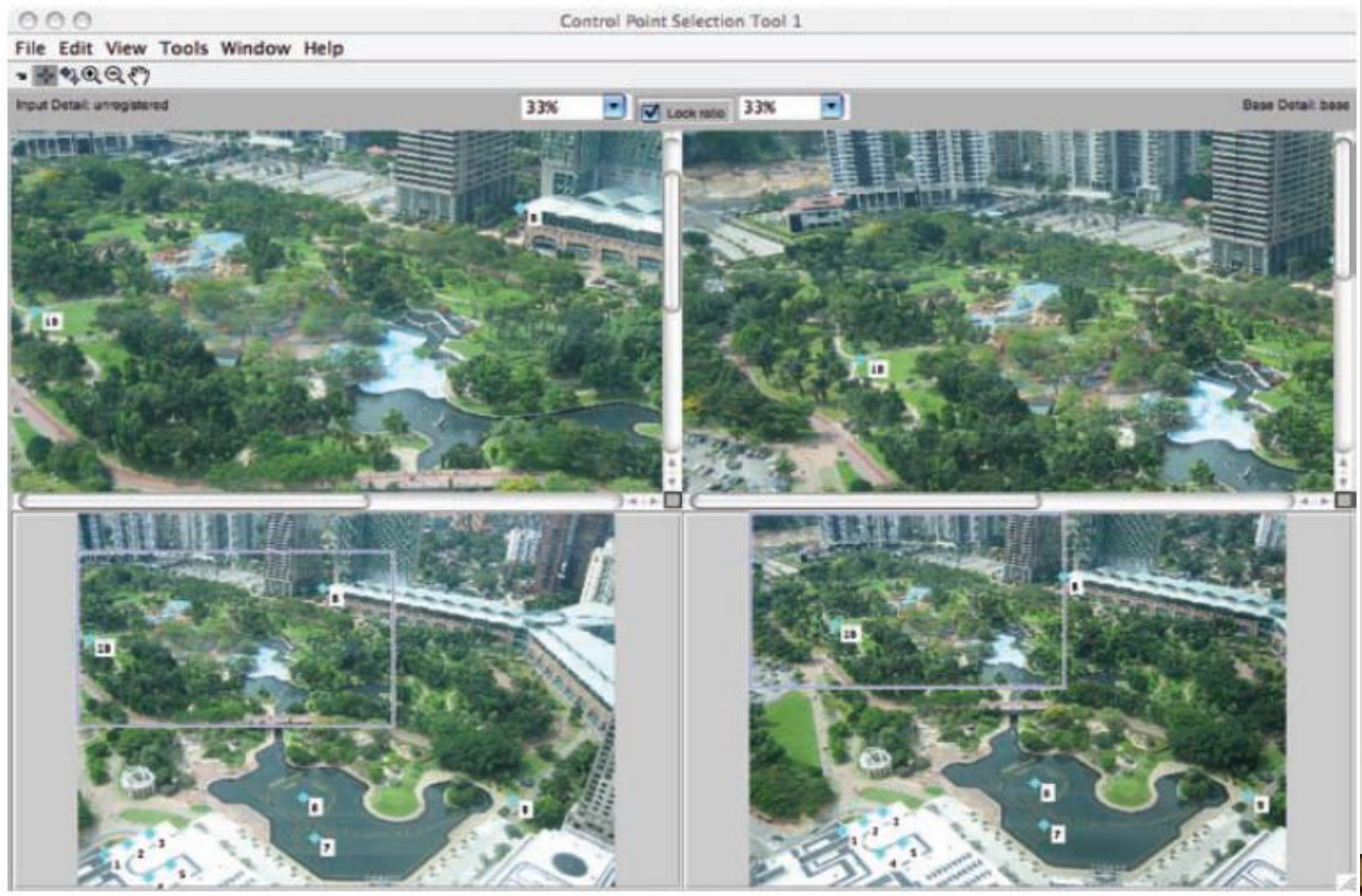

Control Point Selection Tool 1
File Edit View Tools Window Help
Input Detail: unregistered
33%
Lock ratio
33%
Base Detail: base

FIGURE 7.11 Selected points.

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

16. Inspect the coordinates of the selected control points.

```
base_points
input_points
```

17. Use cpcorr to fine-tune the selected control points.

```
input_points_adj = cpcorr(input_points,base_points,...
unregistered(:,:,1),base(:,:,1))
```

Question 4: Compare the values for input_points_adj with that for input_points. Did you notice any changes? Why (not)?

18. This is a critical step. We need to specify the type of transformation we want to apply to the unregistered image based on the type of distortion that it contains. In this case, since the distortion appears to be a combination of translation, rotation, and scaling, we shall use the 'nonreflective similarity' transformation type. This type requires only two pairs of control points.

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

19. Once we have selected the type of transformation, we can determine its parameters using *cp2tform*.
20. Use the resulting *tform* structure to align the unregistered image (using *imtransform*).

```
% Select the type of transformation
        mytform1 = cp2tform(input_points,base_points,...
        'nonreflective similarity');
% Transform the unregistered image
        info = imfinfo('klcc_a.png');
        registered = imtransform(unregistered,mytform1,...
        'XData',[1 info.Width], 'YData',[1 info.Height]);
```

5.11 TUTORIAL: SPATIAL TRANSFORMATIONS AND IMAGE REGISTRATION

21. Display the registered image overlaid on top of the base image.

```
figure, imshow(registered);

hold on;

h = imshow(base);

set(h, 'AlphaData', 0.6);
```

Question 5: Are you happy with the results? If you had to do it again, what would you do differently?

WHAT HAVE WE LEARNED?

- Geometric operations modify *the geometry of an image by repositioning pixels in a constrained way*. They can be used to remove distortions in the image acquisition process or to deliberately introduce a distortion that matches an image with another (e.g., *morphing*).
- *Enlarging or reducing a digital image* can be done with two different purposes in mind: (1) *to actually change the image's dimensions (in pixels)*, which can be accomplished in MATLAB by function *imresize*; (2) *to temporarily change the image size for viewing purposes*, through zooming in/out operations, which can be accomplished in MATLAB as part of the functionality of image display primitives such as *imtool* and *imshow*.

WHAT HAVE WE LEARNED?

- The main interpolation methods used in association with geometric operations are zero-order (or *nearest-neighbor*) interpolation (simple and fast, but leads to low-quality results), first-order (or bilinear) interpolation, and higher-order (e.g., bicubic) interpolation (more sophisticated—and computationally expensive—but leads to best results).
- *Affine transformations* are a special class of geometric operations, such that once applied to an image, straight lines are *preserved, and parallel lines remain parallel*. Translation, rotation, scaling, and shearing are all special cases of affine transformations. MATLAB's IPT has two functions associated with affine transformations: *maketform* and *imtransform*.

WHAT HAVE WE LEARNED?

- Image rotation can be performed using the IPT *imrotate* function.
- An image can be flipped horizontally or vertically in MATLAB using simple linear algebra and matrix manipulation instructions.
- The IPT has a function for cropping images, *imcrop*, which crops an image to a specified rectangle, and which can be specified either interactively (with the mouse) or via parameter passing.
- *Image warping* is a technique by which an image's geometry is changed according to a template.
- *Image morphing* is a geometric transformation technique that converts an image into another in an incremental way. It was popular in TV, movies, and advertisements in the 1980s and 1990s, but has lost impact since then.

PROBLEMS

Problem 8. Use imrotate to rotate an image by an arbitrary angle (not multiple of 90°) using the three interpolation methods discussed in Section 7.3. Compare the visual quality of the results obtained with each method and their computational cost (e.g., using MATLAB functions tic and toc).

Problem 9. Consider the MATLAB snippet below (assume X is a gray-level image) and answer this question: Will X and Z be identical? Explain.

```
Y = imresize(X,0.5,'nearest');
Z = imresize(Y,2.0,'nearest');
```

PROBLEMS

Problem 10. Consider the MATLAB snippet below. It creates an 80 x 80 black-and-white image (B) and uses a simple approach to image interpolation (described in Section 7.3.2) to reduce it to a 40 x 40 pixel equivalent (C). Does the method accomplish its goal? Explain.

```
A = eye(80,80);
A(26:2:54,:)=1;
B = imcomplement(A);
C = B(1:2:end, 1:2:end);
```

END OF LECTURE 5

LECTURE 6 – GRAY-LEVEL TRANSFORMATION, HISTOGRAM AND NEIGHBORHOOD PROCESSING