CHAPTER 6

ARITHMETIC AND LOGIC OPERATIONS

WHAT WILL WE LEARN?

- Which arithmetic and logic operations can be applied to digital images?
- How are they performed in MATLAB?
- What are they used for?

6.1 ARITHMETIC OPERATIONS: FUNDAMENTALS AND APPLICATIONS

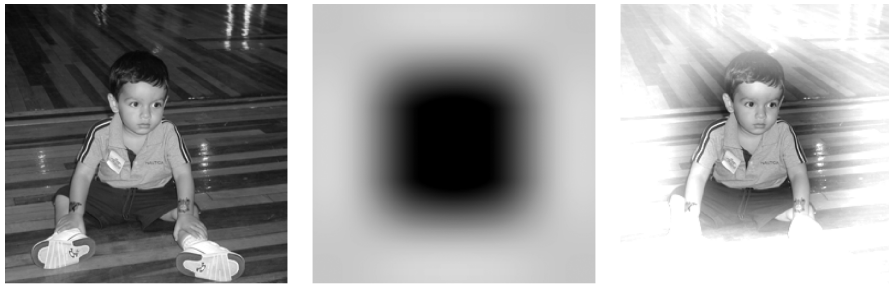
Arithmetic operations involving images are typically performed on a pixel-by-pixel basis; that is, the operation is independently applied to each pixel in the image. Given a 2D array (X) and another 2D array of the same size or a scalar (Y), the resulting array, Z, is obtained by calculating

$$X \quad opn \quad Y = \quad Z \tag{6.1}$$

where *opn* is a binary arithmetic ($+, -, \times, /$) operator.

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

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

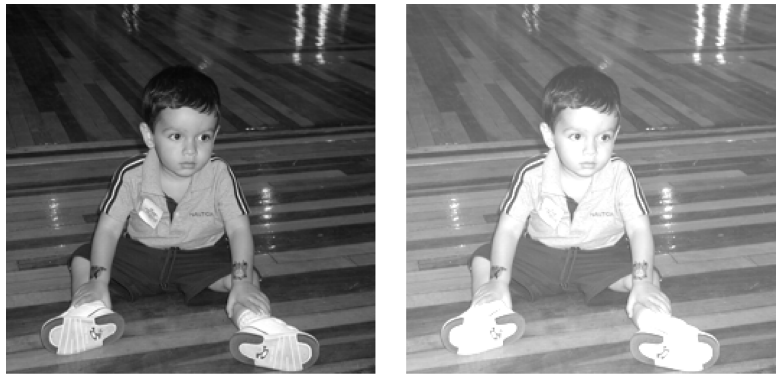


(a) (b) (c)

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

6.1.1 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). 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). 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 such as those described in Chapter 12.



(a) (b)

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

(a) (b) (c)

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

In MATLAB

MATLAB's Image Processing Toolbox (IPT) has a built-in function to add two images or add a constant (scalar) to an image: `imadd`. In Tutorial 6.1 (page 113), you will have a chance to experiment with this function.

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*. Normalization consists in storing the intermediate result in a temporary variable (W) and calculating each resulting pixel value in Z using equation (6.2).

$$g = \frac{L_{\max}}{f_{\max} - f_{\min}}(f - f_{\min}) \tag{6.2}$$

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 6.1

For the two 3 × 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.

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

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

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

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

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

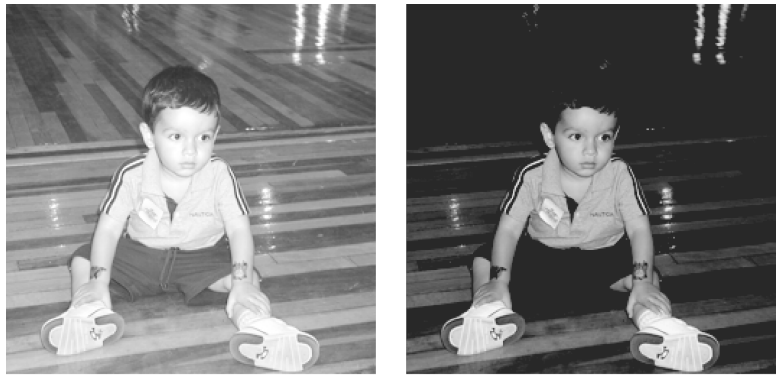
$$Z_{\mathrm{b}} = \begin{bmatrix} 255 & 255 & 255 \\ 45 & 105 & 170 \\ 255 & 255 & 120 \end{bmatrix}$$

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)
```

6.1.2 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



(a) (b)

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

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).

When subtracting one image from another or a constant (scalar) from an image, you 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.

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`. In Tutorial 6.1 (page 113), you will have a chance to experiment with these functions.

■ EXAMPLE 6.2

For the two 3×3 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.

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

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

Solution

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

(a)

$$Z_{\mathrm{a}} = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 150 & 120 \end{bmatrix}$$

(b)

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

(c)

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

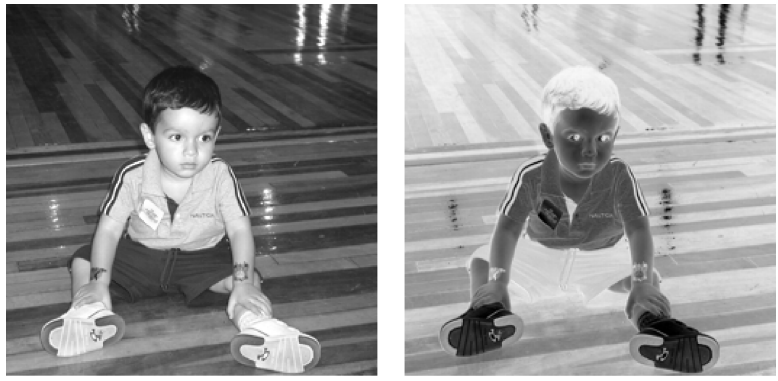
MATLAB code:

```
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 (Figure 6.5):

$$g = -f + L_{\max} \tag{6.3}$$

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.



(a) (b)

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

6.1.3 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.

In MATLAB

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

(a) (b) (c)

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

into another or divide an image by a constant: `imdivide`. In Tutorial 6.1 (page 113), you will have a chance to experiment with these functions.

6.1.4 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 6.3**

Calculate the average of the three 3×3 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`.

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

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

$$Z = \begin{bmatrix} 200 & 160 & 130 \\ 145 & 195 & 120 \\ 105 & 240 & 150 \end{bmatrix}$$

Solution
(a)

$$S_{\mathrm{a}} = \begin{bmatrix} 85 & 85 & 85 \\ 63 & 85 & 85 \\ 85 & 85 & 85 \end{bmatrix}$$

(b)

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

(c)

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

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.

6.2 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.

Figures 6.8–6.11 show examples of AND, OR, XOR, and NOT operations on monochrome images. The AND and OR operations can be used to combine images for special effects purposes. They are also used in masking operations, whose goal is to extract a region of interest (ROI) from an image (see Tutorial 6.2). The XOR operation is often used to highlight differences between two monochrome images. It is, therefore, equivalent to calculating the absolute difference between two images. The NOT operation extracts the binary complement of each pixel value, which is equivalent to applying the "negative" effect on an image.

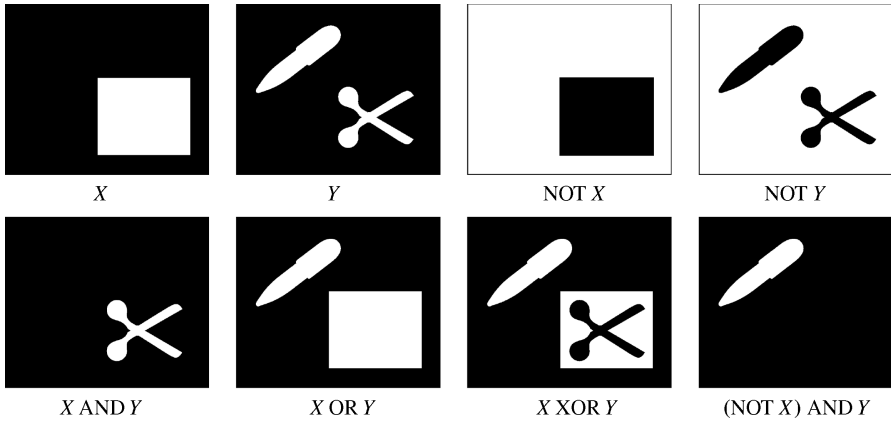


FIGURE 6.7 Logic operations on binary images.

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

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

In MATLAB

MATLAB has built-in functions to perform logic operations on arrays: `bitand`, `bitor`, `bitxor`, and `bitcmp`. In Tutorial 6.2 (page 118), you will have a chance to experiment with these functions.

(a) (b) (c)

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

(a) (b)

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

6.3 TUTORIAL 6.1: 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.

What You Will Need

- `cameraman2.tif`
- `earth1.tif`
- `earth2.tif`
- `gradient.tif`
- `gradient_with_text.tif`

Procedure

The IPT offers four functions to aid in image arithmetic: `imadd`, `imsubtract`, `immultiply`, and `imdivide`. You could use MATLAB's arithmetic functions (`+, −, *, /`) to perform image arithmetic, but it would probably require additional coding to ensure that the operations are performed in double precision, as well as setting cutoff values to be sure that the result is within grayscale range. The functions provided by the IPT do this for you automatically.

Image addition can be used to brighten (or darken) an image by adding (subtracting) a constant value to (from) each pixel value. It can also be used to blend two images into one.

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?

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.
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. 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 you know where the difference is located, you 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. 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');
```

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');
```

```
subplot(2,2,4), imshow(diffim2,[]), ...
     title('Abs Diff Image Scaled');
```

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

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.
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?

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');
```

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')
```

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.
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).

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.

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?

6.4 TUTORIAL 6.2: 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.

What You Will Need

- `lindsay.tif`
- `cameraman2.tif`

Procedure

Logic operators are often used for image masking. We will use the `roipoly` function to create the image mask. Once we have a mask, we will use it to perform logic operations on the selected image.

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`?

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);
```

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. 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)
```

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.

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.

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
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.

- 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`).

6.5 PROBLEMS

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

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

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

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

6.5 Given the 3×3 images X and Y below, obtain (a) X AND Y; (b) X OR Y; (c) X XOR Y.

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

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

6.6 What happens when you add a `uint8` [0, 255] monochrome image to itself?

6.7 What happens when you multiply a `uint8` [0, 255] monochrome image by itself?

6.8 What happens when you multiply a `double` [0, 1.0] monochrome image by itself?

6.9 What happens when you divide a `double` [0, 1.0] monochrome image by itself?

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

6.11 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.

6.12 Write a MATLAB script that reads an image, performs brightness correction using the function written for Problem 6.11, and displays a window with the image and its histogram,[1] before and after the brightness correction operation. What do the histograms tell you? Would you be able to tell from the histograms alone what type of brightness correction was performed? Would you be able to estimate from the histogram information how much brightening or darkening the image experienced?

[1] Histograms will be introduced in Chapter 9, so you may try this problem after reading that chapter.