Point Processing

❑Introduction

❖Any image processing operation transforms the grey values of the pixels.

❖However, image processing operations may be divided into three classes based on the information required to perform the transformation.

❖From the most complex to the simplest, they are:

1. **Transforms.**
   - A transform represents the pixel values in some other, but equivalent form.
   - Transforms allow for some very efficient and powerful algorithms, as we shall see later on.

Point Processing

❑Introduction

- We may consider that in using a transform, the entire image is processed as a single large block.
- This may be illustrated by the diagram shown in figure 2.1.

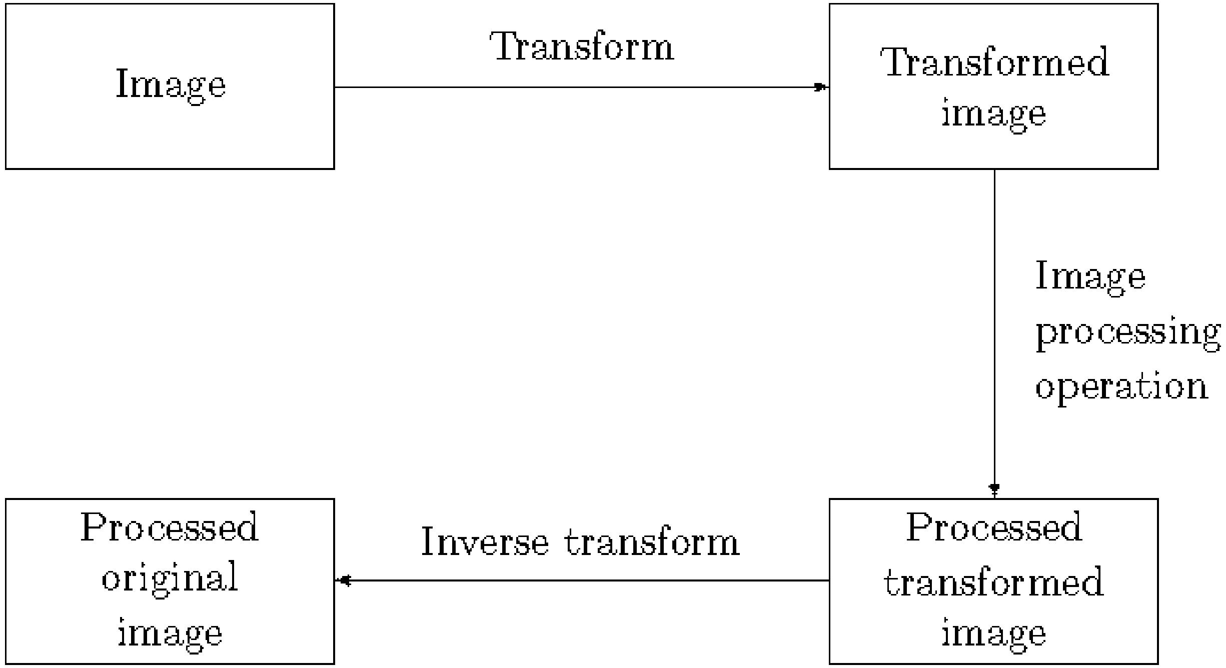


Figure 2.1: Schema for transform processing

Point Processing

❑Introduction

2. **Neighbourhood processing.**
   - To change the grey level of a given pixel we need only know the value of the grey levels in a small neighbourhood of pixels around the given pixel.
3. **Point operations.**
   - A pixel's grey value is changed without any knowledge of its surrounds.

Point Processing

❑Introduction

- Although point operations are the simplest, they contain some of the most powerful and widely used of all image processing operations.
- They are especially useful in image pre-processing, where an image is required to be modified before the main job is attempted.

Point Processing

❑Arithmetic operations

- ❖These operations act by applying a simple function $y = f(x)$ to each grey value in the image.
- ❖Thus $f(x)$ is a function which maps the range $0 \dots 255$ onto itself.
- ❖Simple functions include adding or subtract a constant value to each pixel: $y = x \pm C$
- ❖or multiplying each pixel by a constant: $y = Cx$
- ❖In each case we may have to fiddle the output slightly in order to ensure that the results are integers in the $0 \dots 255$ range.

Point Processing

❑Arithmetic operations

- ❖We can do this by first rounding the result (if necessary) to obtain an integer, and then “clipping” the values by setting:

$$y \leftarrow \begin{cases} 255, & if\ y > 255, \\ 0, & if\ y < 0. \end{cases}$$

- ❖We can obtain an understanding of how these operations affect an image by plotting $y = f(x)$
- ❖Figure 2.2 shows the result of adding or subtracting 128 from each pixel in the image.
- ❖Notice that when we add 128, all grey values of 127 or greater will be mapped to 255.

Point Processing

❑Arithmetic operations

❖And when we subtract 128, all grey values of 128 or less will be mapped to 0.

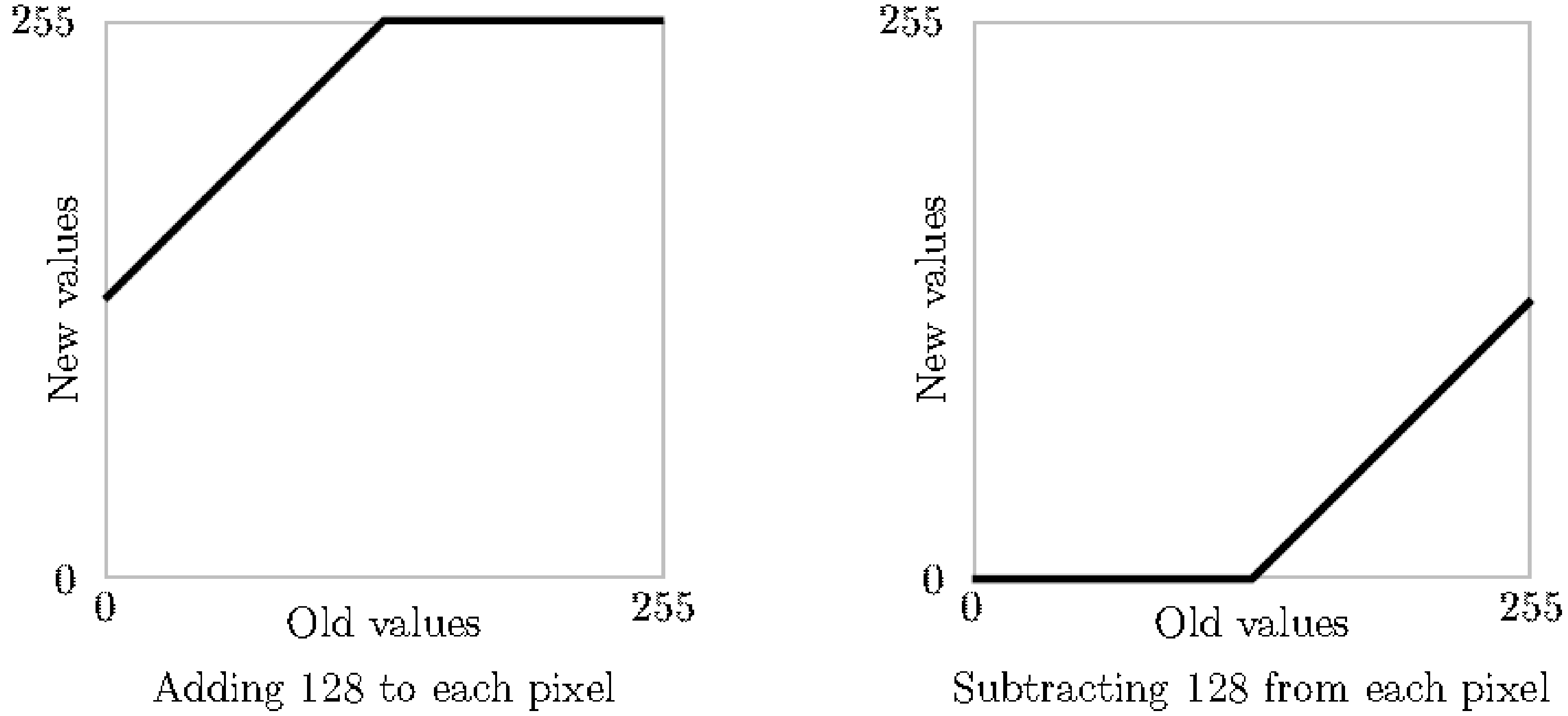


Figure 2.2: Adding and subtracting a constant

Point Processing

❑ Arithmetic operations

- ❖ By looking at these graphs, we observe that in general adding a constant will lighten an image, and subtracting a constant will darken it.
- ❖ We can test this on the blocks image “blocks.tif”, which we have seen in figure 1.4.
- ❖ We start by reading the image in:

```
>> b=imread('blocks.tif');
>> whos b
Name Size Bytes Class
b 256x256 65536 uint8 array
```

Point Processing

❑Arithmetic operations

- ❖The point of the second command was to find the numeric data type of b; it is uint8.
- ❖The unit8 data type is used for data storage only; we can't perform arithmetic operations. If we try, we just get an error message:

```
>> b1=b+128
??? Error using ==> +
Function '+' not defined for variables of class 'uint8'.
```

- ❖We can get round this in two ways.
- ❖We can first turn b into a matrix of type double, add the 128, and then turn back to uint8 for display: `>> b1=uint8(double(b)+128);`

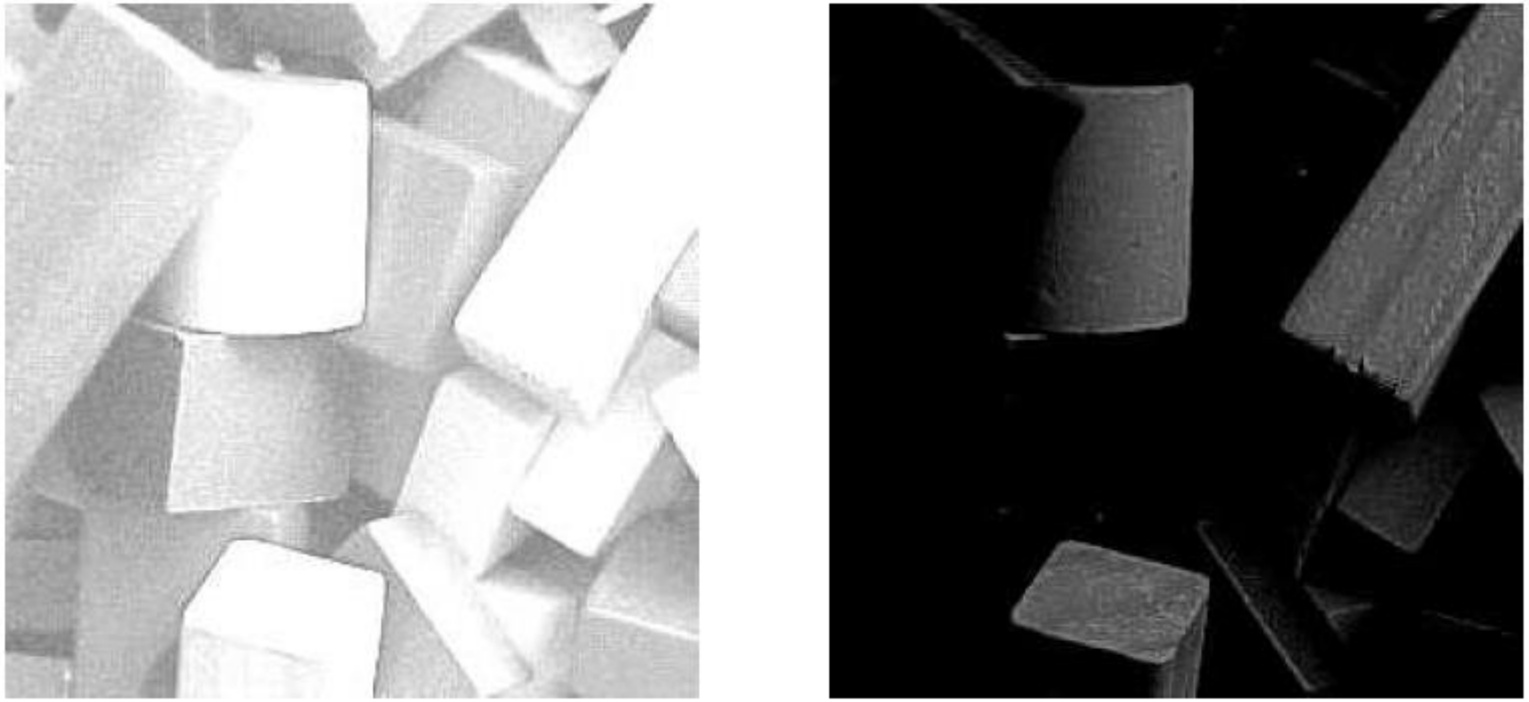
Point Processing

❑Arithmetic operations

- ❖A second, and more elegant way, is to use the Octave function **imadd** which is designed precisely to do this: **>> b1=imadd(b,128);**
- ❖Subtraction is similar; we can transform out matrix in and out of double, or use the **imsubtract** function: **>> b2=imsubtract(b,128);**
- ❖And now we can view them: **>> imshow(b1),figure,imshow(b2)**
- ❖The results are seen in figure 2.3.

Point Processing

❑Arithmetic operations



b1: Adding 128 b2: Subtracting 128

Figure 2.3: Arithmetic operations on an image: adding or subtracting a constant

Point Processing

❑Arithmetic operations

- ❖We can also perform lightening or darkening of an image by multiplication;
- ❖figure 2.4 shows some examples of functions which will have these effects.
- ❖To implement these functions, we use the **immultiply** function.
- ❖Table 2.1 shows the particular commands required to implement the functions of figure 2.4.
- ❖All these images can be viewed with **imshow**;
- ❖they are shown in figure 2.5.
- ❖Compare the results of darkening b2 and b3.

Point Processing

❑Arithmetic operations

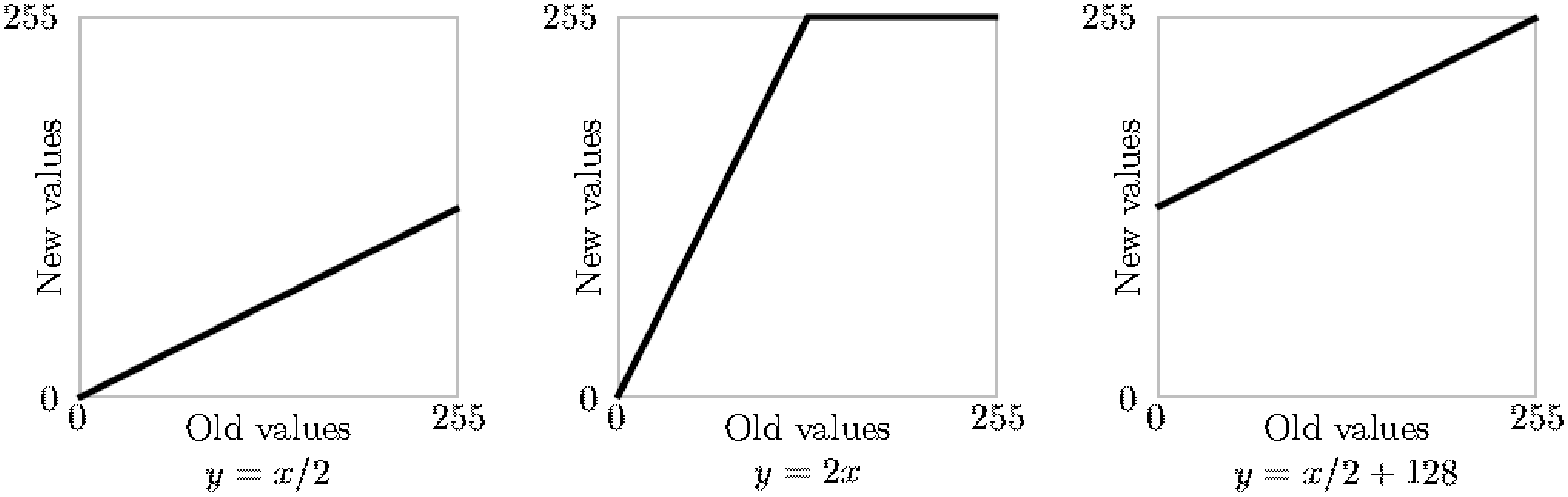


Figure 2.4: Using multiplication and division

$y = x/2$	`b3=immultiply(b,0.5);` or `b3=imdivide(b,2)`
$y = 2x$	`b4=immultiply(b,2);`
$y = x/2 + 128$	`b5=imadd(immultiply(b,0.5),128);` or `b5=imadd(imdivide(b,2),128);`

Table 2.1: Implementing pixel multiplication by MATLAB commands

Point Processing

❑Arithmetic operations

b3: $y = x/2$ b4: $y = 2x$ b5: $y = x/2 + 128$

Figure 2.5: Arithmetic operations on an image: multiplication and division

Point Processing

❑Arithmetic operations

- ❖Note that b3, although darker than the original, is still quite clear, whereas a lot of information has been lost by the subtraction process, as can be seen in image b2.
- ❖This is because in image b2 all pixels with grey values 128 or less have become zero.
- ❖A similar loss of information has occurred in the images b1 and b4.
- ❖Note in particular the edges of the light colored block in the bottom center; in both b1 and b4 the right hand edge has disappeared.
- ❖However, the edge is quite visible in image b5.

Point Processing

❑Arithmetic operations

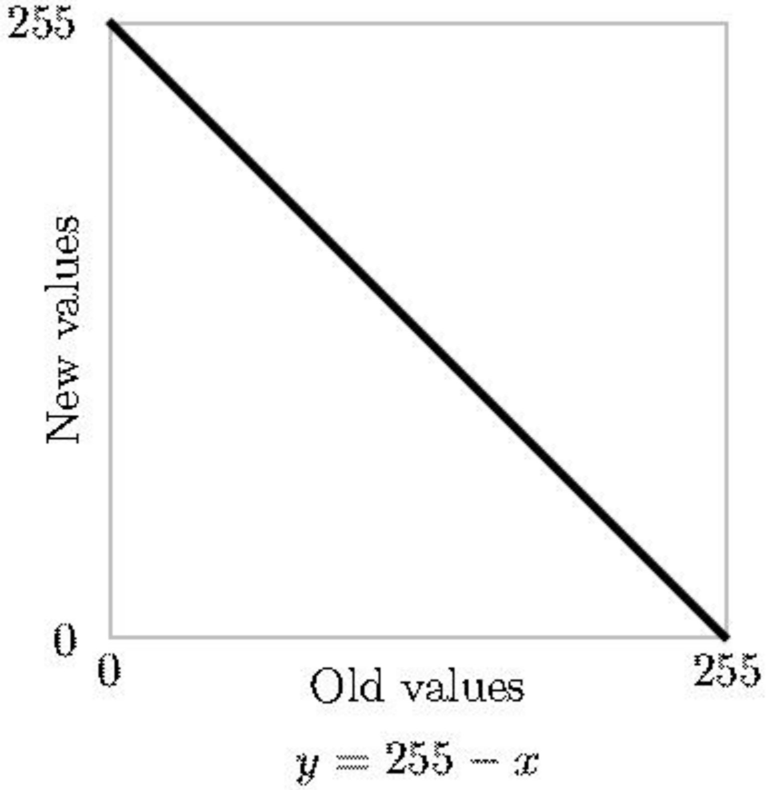
❖Complements

- The complement of a greyscale image is its photographic negative.
- If an image matrix m is of type double and so its grey values are in the range $\mathbf{0.0}$ **to** $\mathbf{1.0}$, we can obtain its negative with the command: **>> 1-m**
- If the image is binary, we can use: **>> ~m**
- If the image is of type uint8, the best approach is the **imcomplement** function.
- Figure 2.6 shows the complement function $y = 255 - x$, and the result of the commands:

>> bc=imcomplement(b), imshow(bc)

Point Processing

❑Arithmetic operations



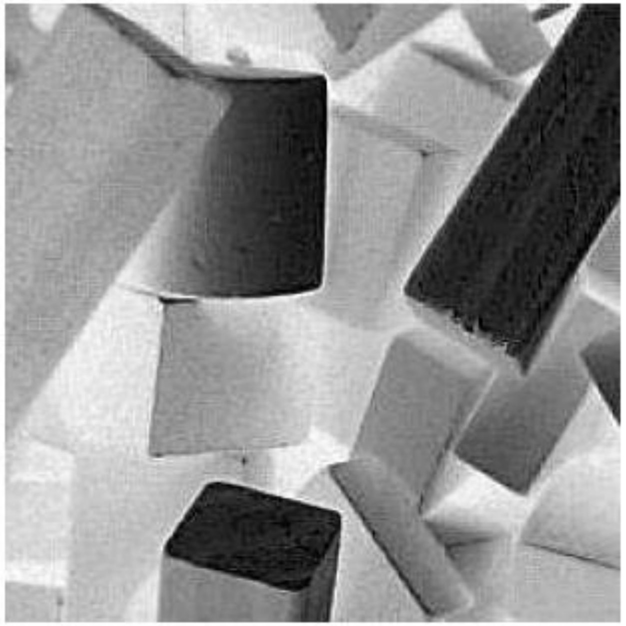


Figure 2.6: Image complementation

Point Processing

❑Arithmetic operations

❖Complements

- Interesting special effects can be obtained by complementing only part of the image;
- for example by taking the complement of pixels of grey value 128 or less, and leaving other pixels untouched.
- Or we could take the complement of pixels which are 128 or greater, and leave other pixels untouched.
- Figure 2.7 shows these functions.
- The effect of these functions is called solarization.

Point Processing

❑Arithmetic operations

❖Complements

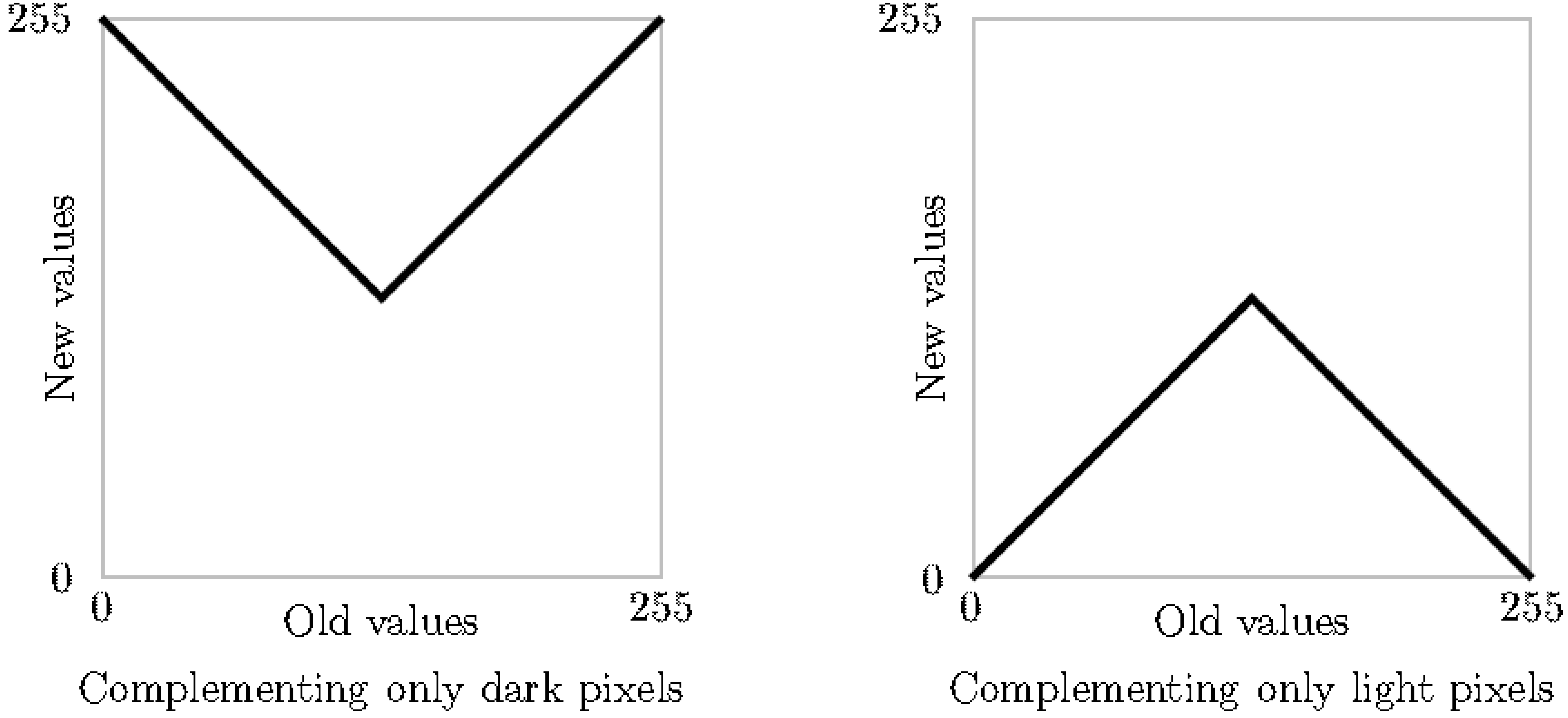


Figure 2.7: Part complementation

Point Processing

❑ Histograms

- ❖ Given a greyscale image, its histogram consists of the histogram of its grey levels;
- ❖ that is, a graph indicating the number of times each grey level occurs in the image.
- ❖ We can infer a great deal about the appearance of an image from its histogram, as the following examples indicate:
  - In a dark image, the grey levels (and hence the histogram) would be clustered at the lower end:
  - In a uniformly bright image, the grey levels would be clustered at the upper end:
  - In a well contrasted image, the grey levels would be well spread out over much of the range:

Point Processing

❑ Histograms

- ❖ We can view the histogram of an image in Octave by using the **imhist** function:

```
>> p=imread('pout.tif');
>> imshow(p),figure,imhist(p),axis tight
```

- ❖ (the axis tight command ensures the axes of the histogram are automatically scaled to fit all the values in).
- ❖ The result is shown in figure 2.8.
- ❖ Since the grey values are all clustered together in the centre of the histogram, we would expect the image to be poorly contrasted, as indeed it is.

Point Processing

❑Histograms

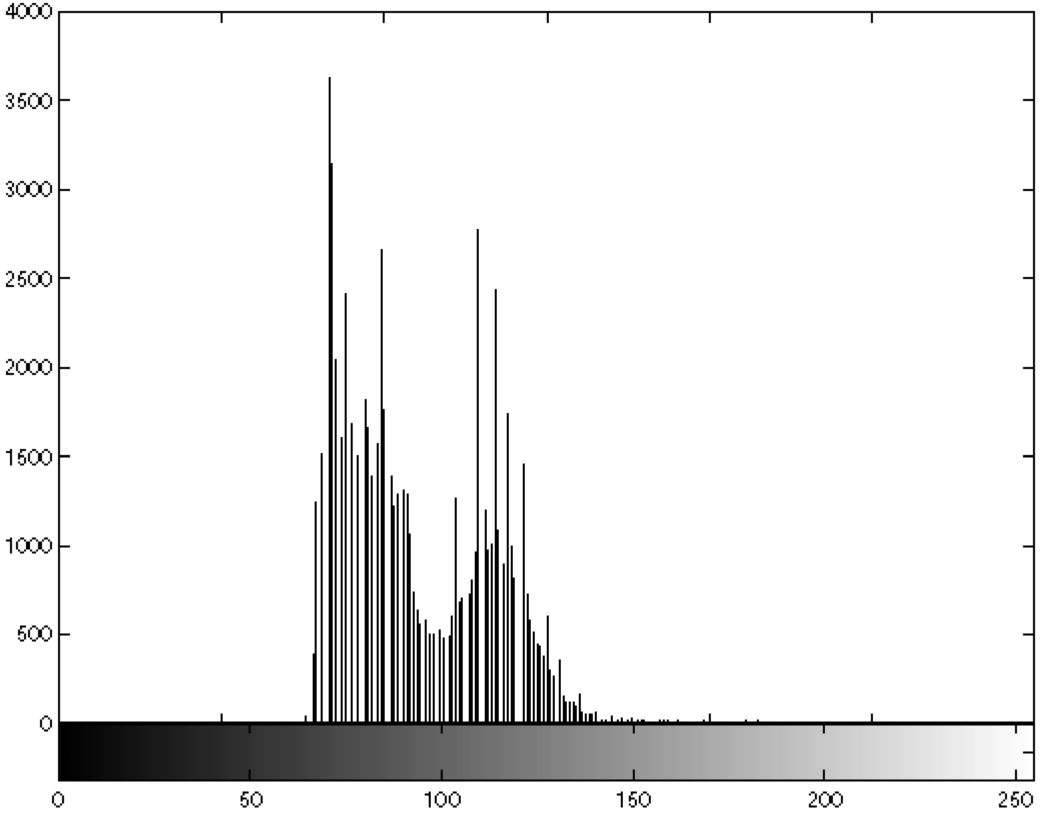


Figure 2.8: The image pout.tif and its histogram

Point Processing

❑Histograms

- ❖Given a poorly contrasted image, we would like to enhance its contrast, by spreading out its histogram.
- ❖There are two ways of doing this.
  1. Histogram stretching (Contrast stretching)
  2. Histogram equalization

Point Processing

❑Histograms

❖Histogram stretching (Contrast stretching)

❖Suppose we have an image with the histogram shown in figure 2.9, associated with a table of the numbers n_i of grey values:

Gray leve i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n_i	15	0	0	0	0	70	110	45	70	35	0	0	0	0	0	15

❖(with n=360 , as before.) We can stretch the grey levels in the centre of the range out by applying the piecewise linear function shown at the right in figure 2.9.

Point Processing

❑Histograms

Gray leve i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
n_i	15	0	0	0	0	70	110	45	70	35	0	0	0	0	0	15

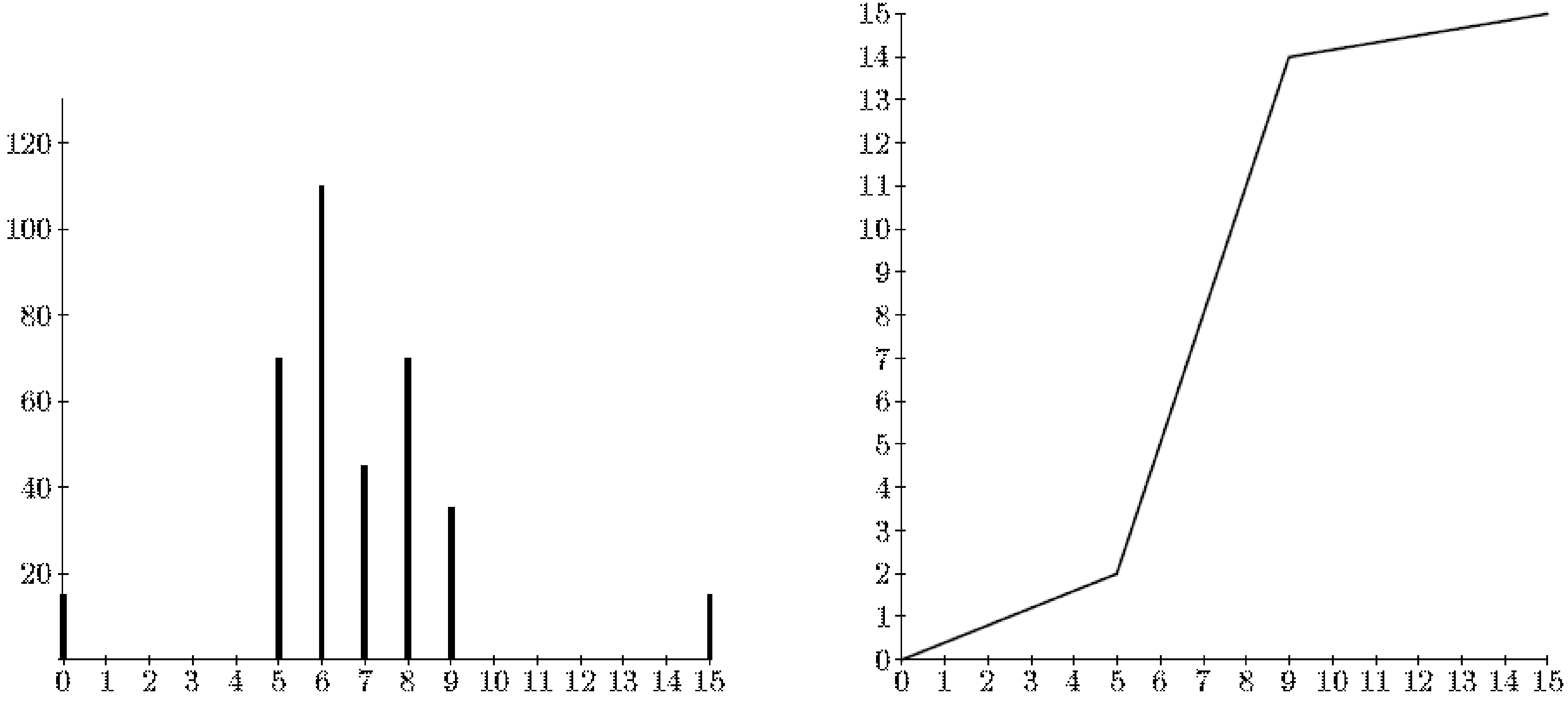


Figure 2.9: A histogram of a poorly contrasted image, and a stretching function

Point Processing

❑Histograms

- ❖This function has the effect of stretching the grey levels 5-9 to grey levels 2-14 according to the equation:

$$j = \frac{14-2}{9-5}(i-5)+2$$

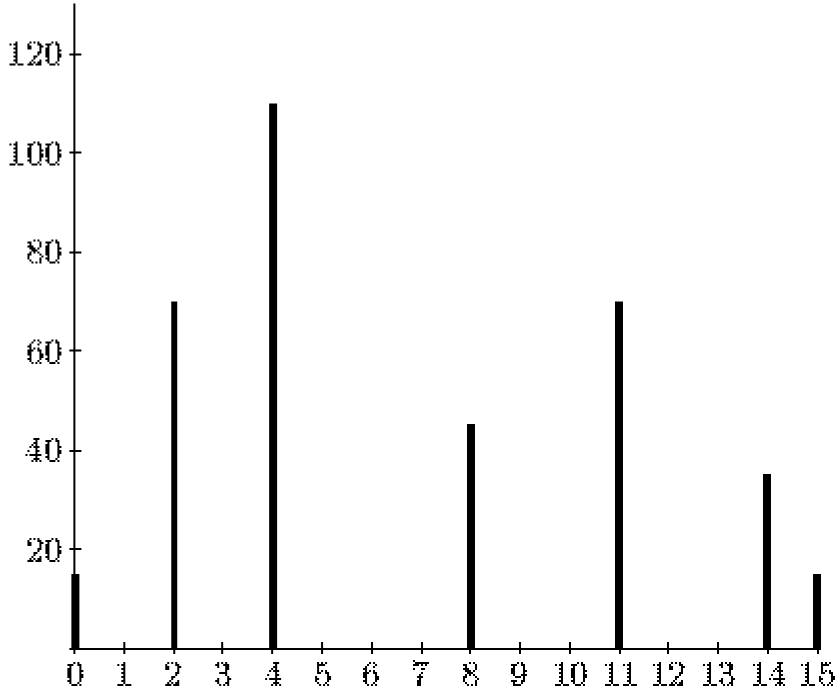
- ❖where i is the original grey level and j its result after the transformation.
- ❖Grey levels outside this range are either left alone (as in this case) or transformed according to the linear functions at the ends of the graph above.
- ❖This yields:

i	5	6	7	8	9
j	2	5	8	11	14

Point Processing

❑Histograms

❖and the corresponding histogram:



❖which indicates an image with greater contrast than the original.

Point Processing

❑Histograms

❖Use of imadjust

❖To perform histogram stretching in Matlab the imadjust function may be used. In its simplest incarnation, the command

>>imadjust(im,[a,b],[c,d])

❖stretches the image according to the function shown in figure 2.10.

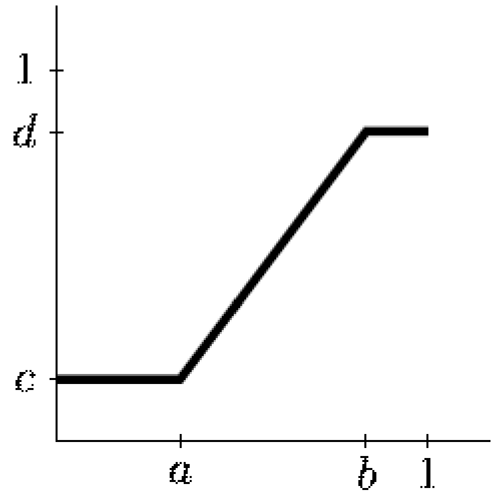


Figure 2.10: The stretching function given by imadjust

Point Processing

❑Histograms

- ❖Use of imadjust
- ❖Since imadjust is designed to work equally well on images of type double, uint8 or uint16 the values of a , b , c and d must be between 0 and 1;
- ❖the function automatically converts the image (if needed) to be of type double.
- ❖Note that imadjust does not work quite in the same way as shown in figure 2.9.
- ❖Pixel values less than a are all converted to c, and pixel values greater than b are all converted to d .
- ❖If either of [a,b] or [c,d] are chosen to be [0,1], the abbreviation [] may be used.

Point Processing

❑Histograms

❖Use of imadjust

- Thus, for example, the command

  >> imadjust(im,[],[])

- does nothing, and the command

  >> imadjust(im,[],[1,0])

- inverts the grey values of the image, to produce a result similar to a photographic negative.
- The imadjust function has one other optional parameter: the gamma value, which describes the shape of the function between the coordinates (a, b) and (c, d).

Point Processing

❑Histograms

❖Use of imadjust

- If gamma is equal to 1, which is the default, then a linear mapping is used, as shown above in figure 2.10.
- However, values less than one produce a function which is concave downward, as shown on the left in figure 2.11, and values greater than one produce a figure which is concave upward, as shown on the right in figure 2.11.

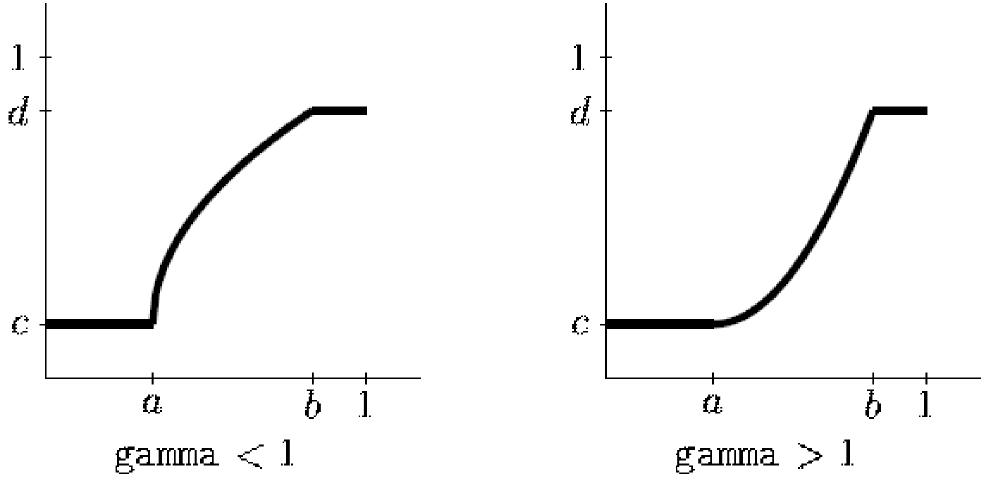


Figure 2.11: The imadjust function with gamma not equal to 1

Point Processing

❑Histograms

❖Use of imadjust

- The function used is a slight variation on the standard line between two points:

$$y = \left(\frac{x-a}{b-a}\right)^{\gamma}(d-c)+c$$

- Use of the gamma value alone can be enough to substantially change the appearance of the image.

Point Processing

❑Histograms

❖Use of imadjust

- For example:

```
>> t=imread('tire.tif');
>> th=imadjust(t,[],[],0.5);
>> imshow(t),figure,imshow(th)
```

- produces the result shown in figure 2.12.

Point Processing

❑Histograms

❖Use of imadjust

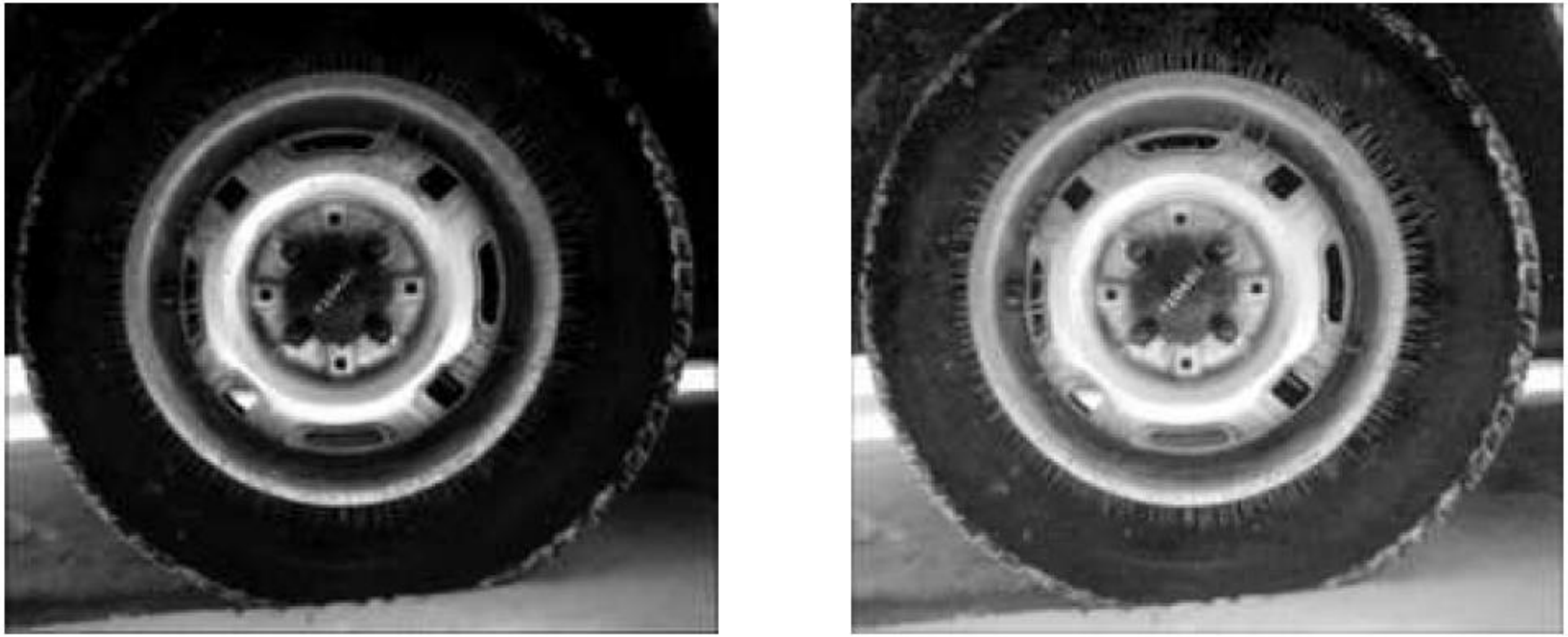


Figure 2.12: The tire image and after adjustment with the gamma value

Point Processing

❑Histograms

❖Use of imadjust

- We may view the imadjust stretching function with the plot function. For example,

  >> plot(t,th,'.'),axis tight

- produces the plot shown in figure 2.13. Since t and th are matrices which contain the original values and the values after the **imadjust** function, the plot function simply plots them, using dots to do it.

Point Processing

❑Histograms

❖Use of imadjust

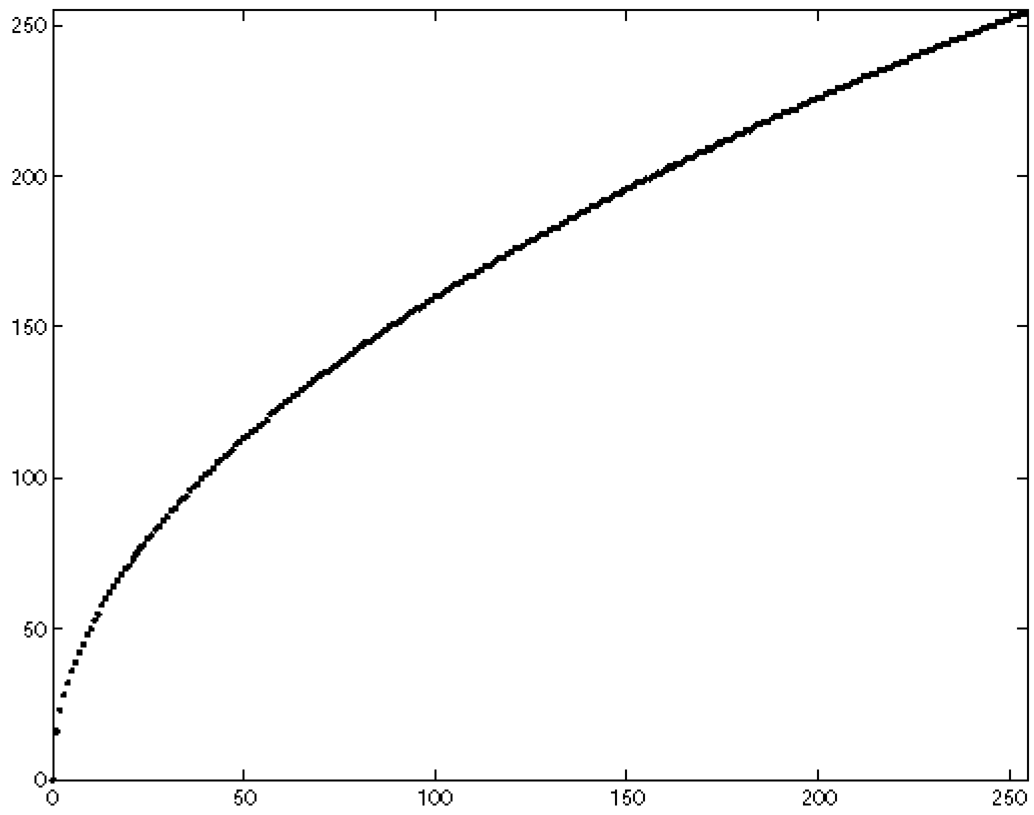


Figure 2.13: The function used in figure 2.12

Point Processing

❑Histograms

- ❖Use of imadjust
- ❖We may view the imadjust stretching function with the plot function. For example,
- ❖>> plot(t,th,'.'),axis tight
- ❖produces the plot shown in gure 2.13. Since p and ph are matrices which contain the original
- ❖values and the values after the imadjust function, the plot function simply plots them, using dots
- ❖to do it.