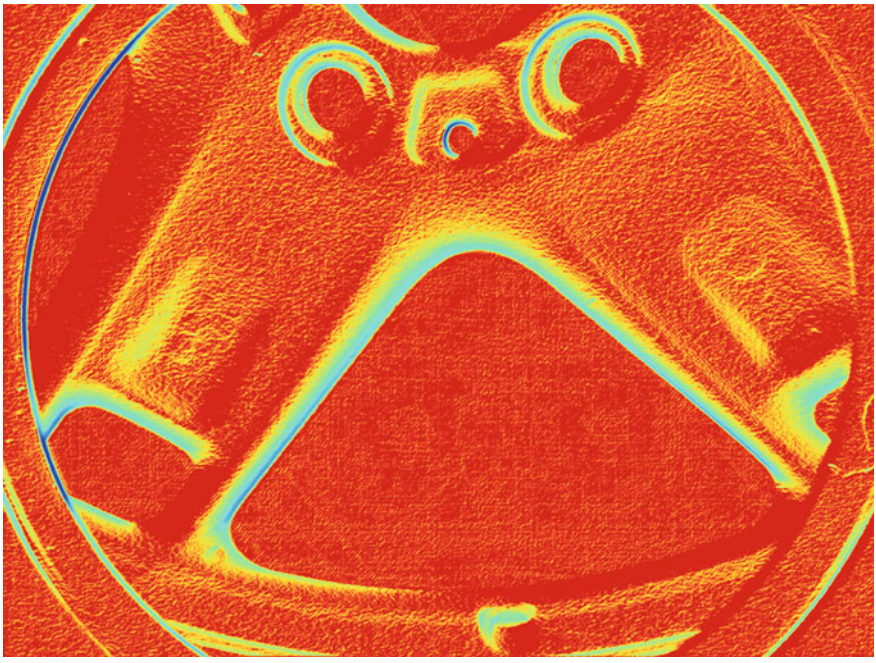
Chapter 4
X-Ray Image Processing

Abstract In this chapter, we cover the main techniques of image processing used in X-ray testing. They are (i) image processing to enhance details, (ii) image filtering to remove noise or detect high-frequency details, (iii) edge detection to identify the boundaries of the objects, (iv) image segmentation to isolate the regions of interest, and (v) to remove the blurriness of the X-ray image. The chapter provides an overview and presents several methodologies with examples using real and simulated X-ray images.



Gradient of an X-ray image of a wheel (from X-ray image `C0001_0001` *colored with 'jet' colormap).*

D. Mery and C. Pieringer, *Computer Vision for X-Ray Testing*,
https://doi.org/10.1007/978-3-030-56769-9_4

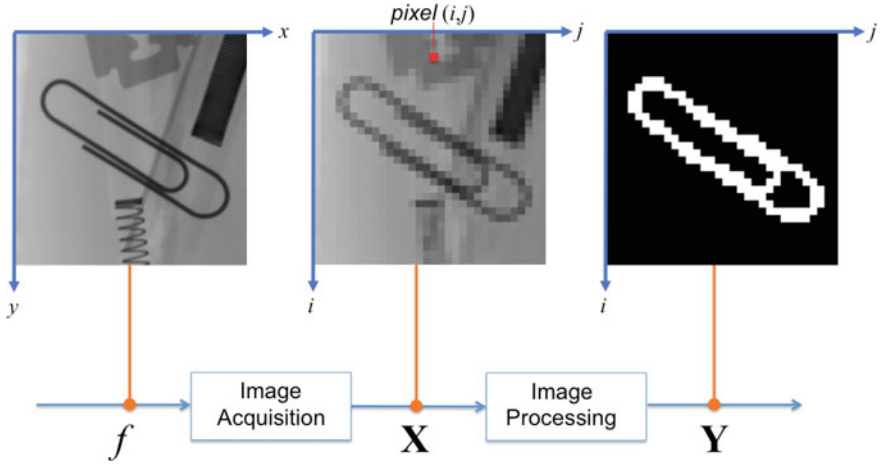


Fig. 4.1 Image processing: input is digital image **X**, output is digital image **Y**

4.1 Introduction

Image processing manipulates a digital image in order to obtain a *new* digital image, i.e., in this process the input is an image and the output is another image. A typical example is *segmentation* as shown in Fig. 4.1, where the input is a grayscale image that contains a clip and the output is a binary image where the pixels that belong to the clip are detected. In our book, we distinguish image processing from image analysis, in which the output is rather an interpretation, a recognition or a measurement of the input image. We will perform image analysis further on, when we learn pattern recognition techniques such as feature extraction (see Sect. 5) and classification (see Sect. 6).

In this chapter, we cover the following image processing techniques that are used in X-ray testing.

- Image preprocessing: The quality of the X-ray image is improved in order to enhance its details.
- Image Filtering: Mainly used to remove noise and detect high-frequency details of the X-ray image.
- Edge detection: The details of the images can be highlighted by detecting the boundaries of the objects of the X-ray image.
- Image segmentation: Regions of interest of the X-ray image are identified and isolated from their surroundings.
- Image restoration: This involves recovering details in blurred images.

In this chapter, we provide an overview of these five techniques. Methodologies and principles will also be outlined, and some application examples followed by limitations to the applicability of the used methodologies will be presented.

In image processing methodology, we have a continuous image f defined in a coordinate system (x, y). Image f is digitalized. The obtained image is a digital image which is stored in matrix $\mathbf{X}$ of size $M \times N$ pixels. The gray value of pixel (i, j) of image $\mathbf{X}$ is $X(i, j)$. Image $\mathbf{X}$ is processed digitally. The output image of this process is image $\mathbf{Y}$, usually a matrix of the same size of $\mathbf{X}$. In this example, the output is a binary image, that means $Y(i, j)$ is '1' (white) and '0' black. Image $\mathbf{Y}$ corresponds to the segmentation of a clip (that is the *object of interest* in this example).

4.2 Image Preprocessing

The X-ray image taken must be preprocessed to improve the quality of the image before it is analyzed. In this section, we will discuss preprocessing techniques that can remove noise, enhance contrast, correct the shading effect, and restore blur deformation in X-ray images.

4.2.1 Noise Removal

Noise in an X-ray image can prove a significant source of image degradation and must be taken into account during image processing and analysis. In an X-ray imaging system, *photon noise* occurs given the quantum nature of X-rays. If we have a system that receives μ photons per pixel in a time ΔT on average, the number of photons striking any particular pixel in any time ΔT will be random. At low levels, however, the noise follows a Poisson law, characterized by the probability:

$$p(x|\mu) = \frac{e^{-\mu}}{\mu^x x!} \tag{4.1}$$

to obtain a value x of photons given its average μ photons in a time ΔT. The standard deviation of this distribution is equal to the square root of the mean.[1] This means that the photon noise amplitude is signal-dependent.

Integration (or averaging) is used to remove X-ray image noise. This technique requires n stationary X-ray images. It computes the filtered image as follows:

$$Y(i, j) = \frac{1}{n} \sum_{k=1}^{n} X_k(i, j), \tag{4.2}$$

[1] At high levels, the Poisson distribution approaches the Gaussian with a standard deviation equal to the square root of the mean: $\sigma = \sqrt{\mu}$.

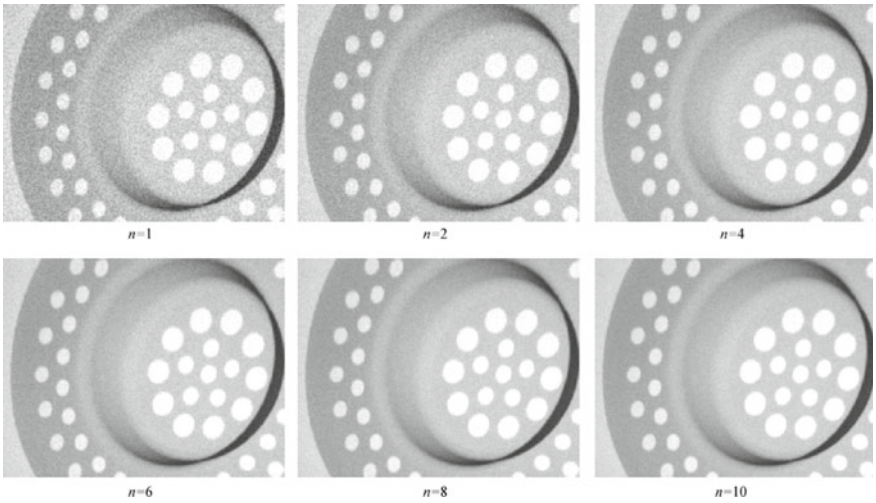


Fig. 4.2 Noise removal after an averaging of n frames. The noise is reduced by factor $\sqrt{n}$

where $X_k(i, j)$ is pixel (i, j) of k-th stationary image and $Y(i, j)$ is the corresponding pixel of the filtered image.

In this technique, the X-ray image noise is modeled using two components: the stationary component (that is constant throughout the n images) and the noise component (that varies from one image to the next). If the noise component has zero mean, by averaging the n images the stationary component is unchanged, while the noise pattern decreases by increasing n. Integrating n stationary X-ray images improves the signal-to-noise ratio by a factor of $\sqrt{n}$ [1, 3].

The effect of image integration is illustrated in Fig. 4.2 that uses n stationary images of an aluminum casting and shows the improvement in the quality of the X-ray image. The larger the number of stationary images n, the better the improvement. Normally, between 10 and 16 stationary images are taken ($10 \leq n \leq 16$).

 Python Example 4.1: In this example, we have 20 noisy X-ray images obtained from a very thin wood piece. The following Python code uses averaging to effectively remove X-ray image noise (4.2):

Listing 4.1 : Noise removal by averaging.

```
import numpy as np
import matplotlib.pylab as plt

from pyxvis.io import gdxraydb

image_set = gdxraydb.Nature()
s = np.double(image_set.load_image(4, 1))

n = 20
for i in range(2, n+1):  # For loops in Python runs until n-1
    xk = np.double(image_set.load_image(4, i))
```

```
    s += xk

y = s / n

fig1, ax = plt.subplots(1, 2, figsize=(16, 8))
ax[0].imshow(s, cmap='gray'), ax[0].axis('off')
ax[1].imshow(y, cmap='gray'), ax[1].axis('off')
plt.show()
```

The output of this code is shown in Fig. 4.3. The reduction of noise is not perfect but very satisfactory. The reader can test this approach on series `C0034` and `C0041` of GDXray+, in which 37 noisy X-ray images of an aluminum wheel with no motion are taken. □

4.2.2 Contrast Enhancement

The gray values in some X-ray images lie in a relatively narrow range of the grayscale. In this case, enhancing the contrast will amplify the differences in the gray values of the image.

We compute the gray value histogram to investigate how an X-ray image uses the grayscale. The function summarizes the gray value information of an X-ray image. The histogram is a function $h(x)$ that denotes the number of pixels in the X-ray image that have a gray value equal to x. Figure 4.4 shows how each histogram represents the distribution of gray values in the X-ray images.

A transformation can be applied to modify the distribution of gray value in an X-ray image. Simple contrast enhancement can be achieved if we use a linear transformation which sets the minimal and maximal gray values of the X-ray image to the minimal and maximal gray value of the grayscale respectively. Thus, the histogram is expanded to occupy the full range of the grayscale. Mathematically, for a scale between 0 and 255, this transformation is expressed as follows:

$$Y(i,j) = 255 \cdot \frac{X(i,j) - x_{\min}}{x_{\max} - x_{\min}}, \tag{4.3}$$

where $x_{\min}$ and $x_{\max}$ denote the minimal and maximal gray value of the input X-ray image. The output image is stored in matrix $\mathbf{Y}$. This simple function is implemented in command linimg from pyxvis Library. Figure 4.4b shows the result of the transformation applied to the X-ray image in Fig. 4.4a. We observe in the histogram of the enhanced X-ray image, how the gray values expand from '0' to '255'. The mapping is linear and means that a gray value equal to $\frac{1}{2}(x_{\max} - x_{\min})$ will be mapped to $255/2$. This linear transformation is illustrated in Fig. 4.5a, where the abscissa is the input gray value and the ordinate is the output gray value.

In a similar fashion, gray input image values can be mapped using a non-linear transformation $y = f(x)$, as illustrated in Fig. 4.5b and c, the results of which are shown in Fig. 4.4c and d respectively. Here, x and y are the gray values of the input and output images respectively. The non-linear transformation is usually performed

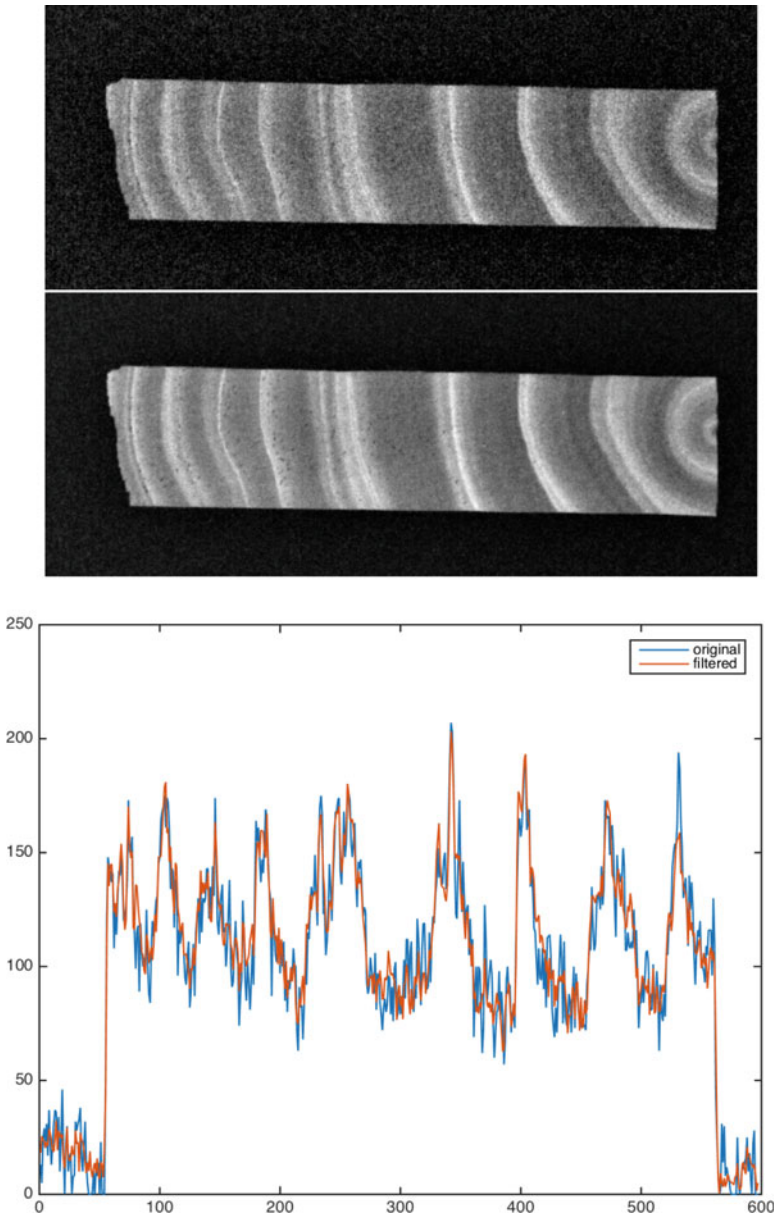


Fig. 4.3 Noise removal of an X-ray image of a wood piece after an averaging of 20 frames. (Top) one of the 20 images. (Middle) filtered image. (Bottom) row 100 of each image. [→ Example 4.1 ]

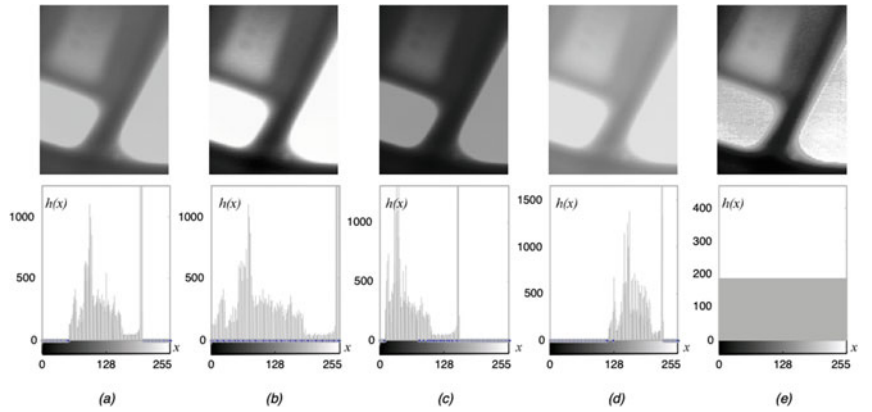


Fig. 4.4 Contrast enhancement: **a** original image, **b** linear transformation ($\gamma = 1$), **c** non-linear transformation ($\gamma = 2$), **d** non-linear transformation ($\gamma = 1/2$), **e** gray values uniformly distributed

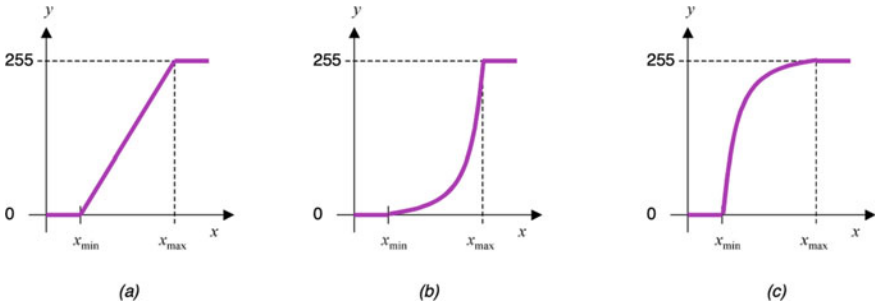


Fig. 4.5 Plots showing different transformations of the gray values: **a** linear transformation ($\gamma = 1$), **b** non-linear transformation with $\gamma > 1$, **c** non-linear transformation with $\gamma < 1$

with a γ *correction* [9]. In these examples, if $\gamma > 1$ the mapping is weighted toward darker output values, and if $\gamma < 1$ the mapping is weighted toward brighter output values. Gamma transformation can be expressed as follows:

$$Y(i,j) = \begin{cases} 0 & \text{for } X(i,j) < x_{\min} \\ 255 \cdot \left[\frac{X(i,j) - x_{\min}}{x_{\max} - x_{\min}} \right]^{\gamma} & \text{for } x_{\min} \leq X(i,j) \leq x_{\max} \\ 255 & \text{for } X(i,j) > x_{\max} \\ . & \end{cases} \tag{4.4}$$

Finally, we present a contrast enhancement equalizing the histogram. Here, we can alter the gray value distribution in order to obtain a desired histogram. A typical equalization corresponds to the uniform histogram as shown in Fig. 4.4d. We see that the number of pixels in the X-ray image for each gray value is constant.

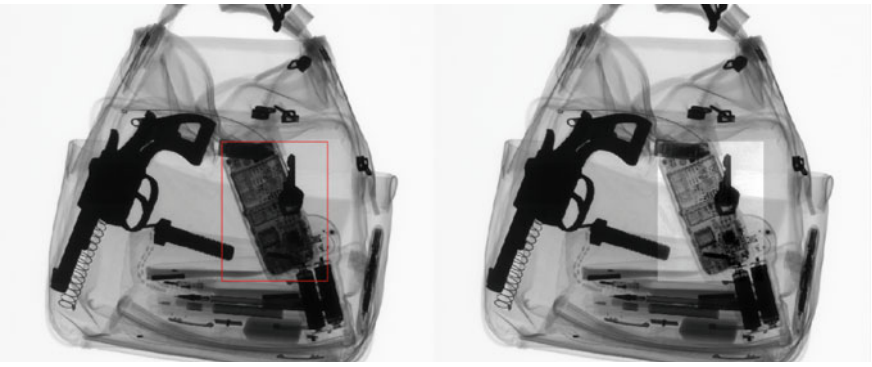


Fig. 4.6 Contrast enhancement by uniforming a histogram of the selected area. [→ Example 4.2]

Phython Example 4.2: In this example, we have an X-ray image of a baggage with very dark zones. The user defines a zone to be enhanced by clicking two opposite corners of a rectangle. The code forces the histogram of this zone to be uniform:

Listing 4.2 : Contrast enhancement of a selected area.

```
import numpy as np
import matplotlib.pylab as plt

from pyxvis.io import gdxraydb
from pyxvis.processing.images import hist_forceuni

image_set = gdxraydb.Baggages()
img = np.double(image_set.load_image(44, 130))

x_box = img[750:2000, 1250:2000]
x_box = hist_forceuni(x_box)
img2 = img.copy()
img2[750:2000, 1250:2000] = x_box

fig1, ax = plt.subplots(1, 2, figsize=(16, 8))
ax[0].imshow(img, cmap='gray'), ax[0].axis('off')
ax[1].imshow(img2, cmap='gray'), ax[1].axis('off')
plt.show()
```

The output of this code is shown in Fig. 4.6. For the equalization, the code uses function hist_forceuni from pyxvis Library. □

4.2.3 Shading Correction

A decrease in the angular intensity in the projection of the X-rays causes low spatial frequency variations in X-ray images [1, 7]. An example is illustrated in Fig. 4.7a,

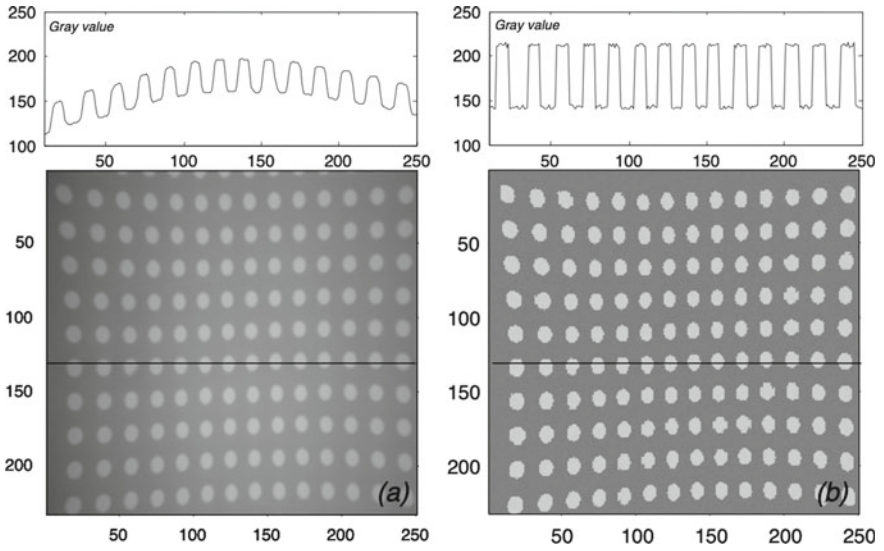


Fig. 4.7 Shading correction: **a** original image, **b** image after shading correction. The corresponding gray values profiles of row number 130 are shown above the images

which shows an X-ray image of an aluminum plate with holes in it. Since the plate is of a constant thickness, we would expect to see a constant gray value for the aluminum part and another constant gray value for the holes. In fact, the X-ray image is darker at the corners. This deficiency can be overcome by using a linear shading correction.

In this technique, we take two images as shown in Fig. 4.8. The first one, $\mathbf{r}_1$, of a thin plate, and the second one, $\mathbf{r}_2$, of a thick plate. We define i_1 and i_2 as the ideal gray values for the first and second images respectively. From $\mathbf{r}_1$, $\mathbf{r}_2$, i_1, and i_2, offset and gain, correction matrices **a** and **b** are calculated assuming a linear transformation between the original X-ray image **x** and corrected X-ray image **y**:

$$Y(i, j) = a(i, j) \cdot X(i, j) + b(i, j), \tag{4.5}$$

where the offset and gain matrices are computed as follows:

$$a(i, j) = \frac{i_2 - i_1}{r_2(i, j) - r_1(i, j)} \qquad b(i, j) = i_1 - r_1(i, j) \cdot a(i, j). \tag{4.6}$$

An example of this technique is illustrated in Fig. 4.7b. In this case, we obtain only two gray values (with noise) one for the aluminum part and another for the holes of the plate.


Python Example 4.3: In this example, we simulate images **X** (a plate with a square cavity). In addition, we simulate X-ray images $\mathbf{r}_1$ (a thin plate) and $\mathbf{r}_2$ (a thick

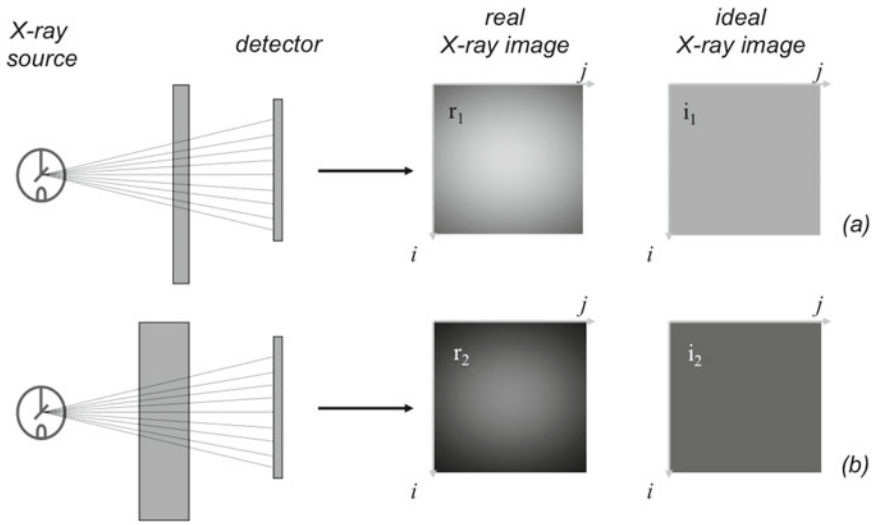


Fig. 4.8 Shading correction: **a** X-ray image for a thin plate, **b** X-ray image for a thick plate. Ideal X-ray images have a constant gray value

plate) as illustrated in Fig. 4.8. The following Python code shows how the shading effect of **X** can be corrected:

Listing 4.3 : Shading correction.

```
import numpy as np
import matplotlib.pylab as plt

from pyxvis.processing.images import shading, fspecial

mat_r1 = fspecial('gaussian', 256, 80)
mat_r1 = mat_r1 / np.max(mat_r1.flatten()) * 0.8

mat_r2 = fspecial('gaussian', 256, 60)
mat_r2 = mat_r2 / np.max(mat_r2.flatten()) * 0.4

i1 = 0.8
i2 = 0.4

mat_x = fspecial('gaussian', 256, 70)
mat_x = mat_x / np.max(mat_x.flatten()) * 0.7
mat_x[30:80, 30:80] = mat_x[30:80, 30:80] * 1.5

mat_y = shading(mat_x, mat_r1, mat_r2, i1, i2)

fig, ax = plt.subplots(1, 2, figsize=(10, 10))
ax[0].imshow(mat_x, cmap='gray')
ax[0].axis('off');
ax[1].imshow(mat_y, cmap='gray')
ax[1].axis('off');

plt.show()
```

The output of this code is shown in Fig. 4.9. The correction is evident: the appearance of the background is homogenous, whereas the square is more distinguishable. In this code, we use function shading of pyxvis Library. This function computes shading correction as defined in (4.5). □

4.3 Image Filtering

2D image filtering is performed in digital image processing using a small neighborhood of a pixel $X(i, j)$ in an input image to produce a new gray value $Y(i, j)$ in the output image, as shown in Fig. 4.10. A *filter mask* defines the input pixels to be processed by an *operator* f. The resulting value is the output pixel. The output for the entire image is obtained by shifting the mask over the input image. Mathematically, the image filtering is expressed as follows:

$$Y(i, j) = f[\underbrace{X(i - p, j - p), \ldots, X(i, j), \ldots, X(i + p, j + p)}_{\text{input pixels}}], \tag{4.7}$$

for $i = p + 1 \ldots M - p$ and $j = p + 1 \ldots N - p$, where M and N are the number of rows and columns of the input and output images. The size of the filter mask is, in this case, $(2p + 1) \times (2p + 1)$. The operator f can be linear or non-linear. In this section, the most important linear and non-linear filters for X-ray testing are outlined.

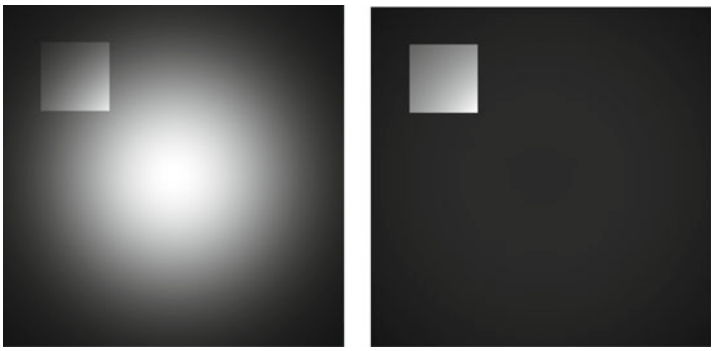


Fig. 4.9 Simulation of shading correction: (Left) X-ray image for a plate with a square cavity (**X**), (Right) corrected image (**Y**). [→ Example 4.3]

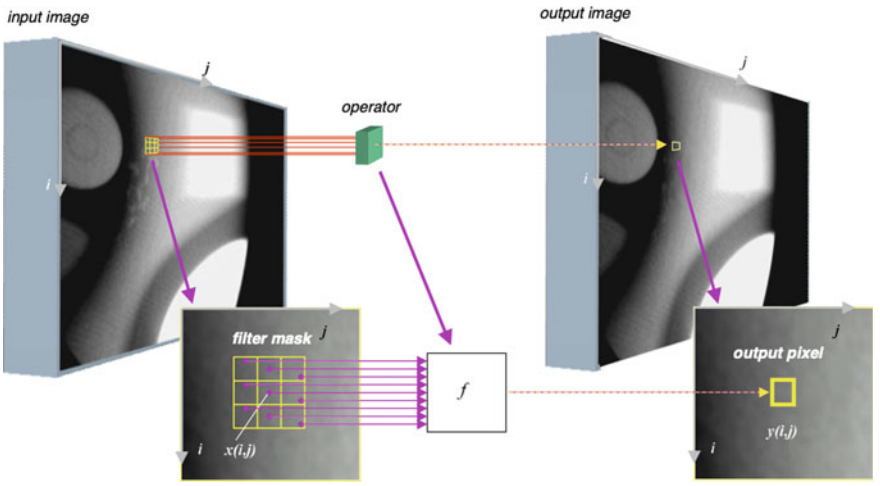


Fig. 4.10 Image filtering

4.3.1 Linear Filtering

The operator f is linear, if the resulting value $Y(i, j)$ is calculated as a linear combination of the input pixels:

$$Y(i, j) = \sum_{m=-p}^{p} \sum_{n=-p}^{p} h(m, n) \cdot X(i - m, j - n), \tag{4.8}$$

where **h** is called the *convolution mask*. The elements of **h** weight the input pixels. The convolution of an image **X** with a mask **h** can be written as follows:

$$\mathbf{Y} = \mathbf{X} * \mathbf{h}. \tag{4.9}$$

Averaging is a simple example of linear filtering. For a 3×3 neighborhood, the convolution mask is

$$\mathbf{h} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Gaussian mask can be used as well

$$h(m, n) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{m^2+n^2}{2\sigma^2}} \tag{4.10}$$

scale factor $1/(2\pi\sigma^2)$ ensures $\sum_{m,n} h(m,n) = 1$ over all elements of $\mathbf{h}$. Average and Gaussian filtering are implemented respectively as functions im_average and im_gaussian in pyxvis Library.

A common application of filtering in X-ray testing is defect detection, e.g., in castings and welds. Filtering out defects detected in an X-ray image will provide a reference *defect-free* image. The defects are detected by finding deviations in the original image from the reference image. The problem is how one can generate a defect-free image from the original X-ray image. Assuming that the defects will be smaller than the regular structure of the test piece, one can use a low-pass filter that does not consider the high-frequency components of the image. However, if a linear filter is used for this task, the edges of the regular structure of the specimen are not necessarily preserved and many false alarms are raised at the edges of regular structures. Consequently, a non-linear filter is used.

4.3.2 Non-linear Filtering

In order to avoid the mentioned problems of linear filters, non-linear filters are used. Defect discrimination can be performed with a *median filter*. The median filter is a ranking operator (and thus non-linear), where the output value is the middle value of the input values ordered in a rising sequence [5]. For an even number of input numbers, the median value is the arithmetic mean of the two middle values.

The application of a median filter is useful for generating the reference image because it smoothes noise yet preserves sharp edges, whereas other linear low-pass filters blur such edges (see a comparison with linear filters in Fig. 4.11). Hence, it follows that small defects can be suppressed, while the regular structures are preserved. Figure 4.12 shows this phenomenon for a 1D example. The input signal x is filtered using a median filter with nine input elements, and the resulting signal is y. We can see that structures of length n greater than four cannot be eliminated. The third column shows the detection $x - y$. Large structures of $n \geq 5$ are not detected as presented in the last two cases.

If the background captured by the median filter is constant, foreground structures could be suppressed if the number of values belonging to the structure is less than one half of the input value to the filter. This characteristic is utilized to suppress the defect structures and to preserve the design features of the test piece in the image.

An example for the application of a median filter on 2D signals (images) is shown in Fig. 4.13 and includes different structures and mask sizes compared to the effects of two linear low-pass filters. One can appreciate that only the median filter manages to suppress the relatively small structures completely, whereas the large patterns retain their gray values and sharp edges.

The goal of the background image function, therefore, is to create a defect-free image from the test image. A real example is shown in Fig. 4.14. In this example, from an original X-ray image $\mathbf{X}$, we generate a filtered image $\mathbf{Y}$ and a difference image $|\mathbf{X} - \mathbf{Y}|$. By setting a threshold, we obtain a binary image whose pixels are

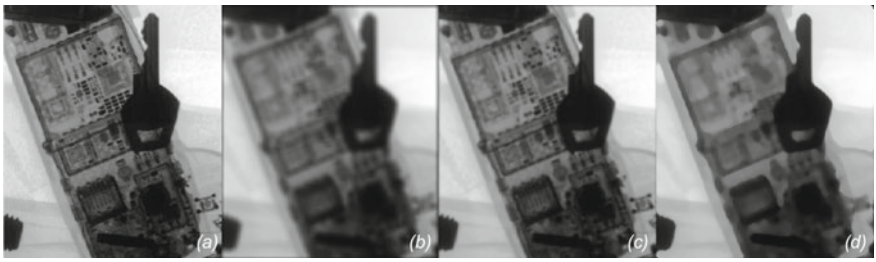


Fig. 4.11 Example of filtering of **a** an X-ray image of 600 × 700 pixels using **b** arithmetic, **c** Gaussian and **d** median filters with a mask of 19 × 19 pixels. The filtered images where obtained using commands im_average, im_gaussian, and im_median of pyxvis Library

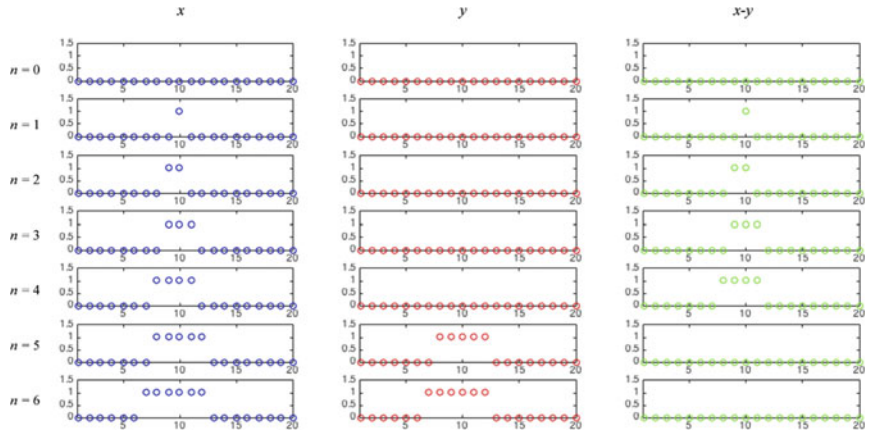


Fig. 4.12 Median filter application on a 1D signal x. The filtered signal is y (the size of the median mask is 9). Structures of length n less than 9/2 are eliminated in y. This filter can be used to detect small structures ($n \leq 4$)

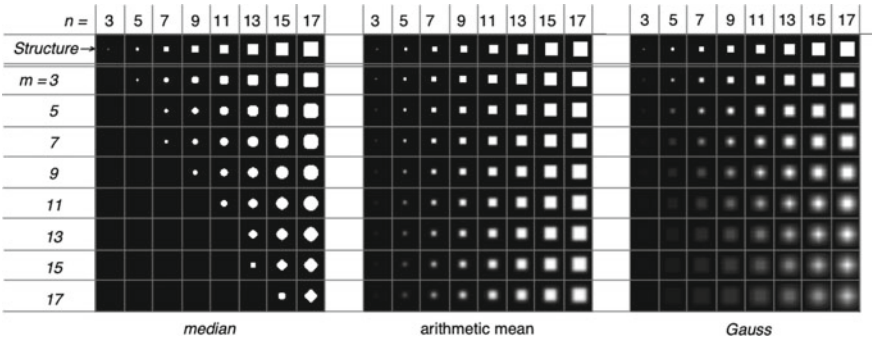


Fig. 4.13 Median filter application on an $n \times n$ structure using an $m \times m$ quadratic mask compared to average and Gauss low-pass filter application

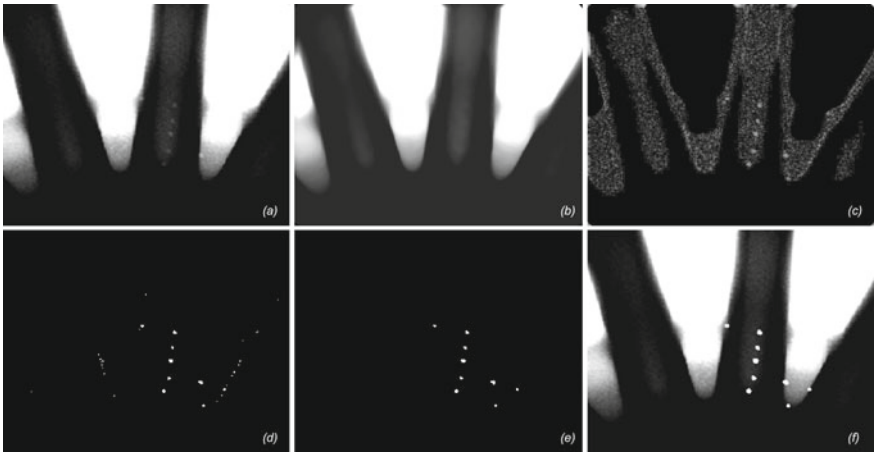


Fig. 4.14 Defect detection using median filtering: **a** original X-ray image of an aluminum wheel with small defects, **b** filtered X-ray image, **c** difference image, **d** binary image using a threshold, **e** elimination of small regions, **f** detection superimposed onto original image. [→ Example 4.4]

'1' (or white), where the gray values in the difference image are greater than the selected threshold. Finally, we eliminate very small regions. The remaining pixels correspond to the detected flaws.

Python Example 4.4: In this example, we detect small defects of an aluminum wheel. First, a reference defect-free image is estimated from the original image itself using median filtering. Second, the difference between original and reference images is computed. Finally, defects are detected when the difference in gray values is high enough and the size of the detected region is large enough:

Listing 4.4 : Defect detection using median filtering

```
import numpy as np
import matplotlib.pylab as plt

from skimage.morphology import remove_small_objects, binary_dilation
from skimage.segmentation import clear_border

from pyxvis.io import gdxraydb
from pyxvis.io.visualization import binview
from pyxvis.processing.images import im_gaussian, im_median

image_set = gdxraydb.Castings()

X = image_set.load_image(21, 29)  # Original image
X = im_gaussian(X, k=5)  # Low pass filtering

fig1, ax1 = plt.subplots(1, 1, figsize=(6, 6))
ax1.set_title('Original Image with defects')
ax1.imshow(X, cmap='gray')
ax1.axis('off')
plt.show()
```

```
Y0 = im_median(X, k=23)
fig2, ax2 = plt.subplots(1, 1, figsize=(6, 6))
ax2.set_title('Median filter')
ax2.imshow(Y0, cmap='gray')
ax2.axis('off')
plt.show()

Y1 = np.abs(np.double(X) - np.double(Y0))
fig3, ax3 = plt.subplots(1, 1, figsize=(6, 6))
ax3.set_title('Difference Image')
ax3.imshow(np.log10(Y1 + 1), cmap='gray')
ax3.axis('off')
plt.show()

Y2 = Y1 > 18
fig4, ax4 = plt.subplots(1, 1, figsize=(6, 6))
ax4.set_title('Binary')
ax4.imshow(Y2, cmap='gray')
ax4.axis('off')
plt.show()

Y3 = remove_small_objects(Y2, 20)
fig5, ax5 = plt.subplots(1, 1, figsize=(6, 6))
ax5.set_title('Binary')
ax5.imshow(Y3, cmap='gray')
ax5.axis('off')
plt.show()

Y = clear_border(binary_dilation(Y3, np.ones((3, 3))))
fig6, ax6 = plt.subplots(1, 1, figsize=(6, 6))
ax6.set_title('Small region are eliminated')
ax6.imshow(Y, cmap='gray')
ax6.axis('off')
plt.show()

blend_mask = binview(X, Y, 'y', 1)
fig6, ax6 = plt.subplots(1, 1, figsize=(6, 6))
ax6.set_title('Small region are eliminated')
ax6.imshow(blend_mask, cmap='gray')
ax6.axis('off')
plt.show()
```

The output of this code—step by step—is shown in the last row of Fig. 4.14. □

4.4 Edge Detection

In this section, we will study how the *edges* of an X-ray image can be detected. The edges correspond to pixels of the image in which the gray value changes significantly over a short distance [3]. Since edges are discontinuities in the intensity of the X-ray image, they are normally estimated by maximizing the gradient of the image. Edge detection image corresponds to a binary image (of the same size of the X-ray image), where a pixel is '1' if it belongs to an edge; otherwise, it is '0', as shown in Fig. 4.15. Before we begin a more detailed description of edge detection, it is worthwhile to highlight some aspects of its relevance in the analysis of X-ray images.

The edges of an X-ray image should show the boundary of objects, e.g., boundaries of defects in control quality of aluminum castings, boundaries of the weld

Fig. 4.15 Edge detection of an X-ray image of a pen case. The edges correspond the boundaries of the objects that are inside the pen case. [→ Example 4.11]

in welding inspection and boundaries of objects in baggage screening (Fig. 4.15). Thus, the input X-ray image is transformed into a binary image which shows structural properties of the X-ray image. The key idea is to detect objects of interest, such as defects in case of quality control or threatening objects in case of baggage screening, based on the information provided by edge detection.

In this section, we will review some basic edge detection techniques that have been used in X-ray testing: gradient estimation (Sect. 4.4.1), LoG, Laplacian-of-Gaussian (Sect. 4.4.2), and Canny (Sect. 4.4.3). Segmentation techniques based on edge detection will be outlined in Sect. 4.5.

4.4.1 Gradient Estimation

The gradient for a 1D function $f(x)$ is defined by

$$f'(x) = \frac{\partial f}{\partial x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{4.11}$$

and for a 2D function $f(x, y)$ is defined by a vector of two elements, one in x direction, and the another one in y direction:

$$\nabla f(x, y) = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right]. \tag{4.12}$$

In digital images, after digitalization of $f(x, y)$, however, corresponding Δx or Δy values cannot be less than one pixel. A simple way to compute the gradient of image $\mathbf{X}$ in i and j direction can be respectively:

$$G_i(i, j) = X(i + 1, j) - X(i, j) \quad \text{and} \quad G_j(i, j) = X(i, j + 1) - X(i, j). \tag{4.13}$$

Thus, the magnitude of the gradient can be computed as follows:

$$G(i, j) = \sqrt{(G_i(i, j))^2 + (G_j(i, j))^2} \tag{4.14}$$

and the direction of the gradient as follows:

$$A(i,j) = \arctan \frac{G_j(i,j)}{G_i(i,j)}. \tag{4.15}$$

In this formulation, gradient images $\mathbf{G}_i$ and $\mathbf{G}_j$ can be easily calculated by convolution (4.9). Thus,

$$\mathbf{G}_i = \mathbf{X} * \mathbf{h}^{\mathsf{T}} \quad \text{and} \quad \mathbf{G}_j = \mathbf{X} * \mathbf{h}, \tag{4.16}$$

where $\mathbf{h}$ is the mask used to compute the gradient in horizontal direction. For instance, if we compute the gradient using the simple way (4.13), we can use $\mathbf{h} = [-1 \ +1]$ in (4.16). Nevertheless, for noisy images, larger masks are suggested for (4.16). Sobel and Prewitt masks are commonly used in image processing [5]. They are defined as follows:

$$\mathbf{h}_{\text{Sobel}} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad \text{and} \quad \mathbf{h}_{\text{Prewitt}} = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix}. \tag{4.17}$$

For severe noise, it is recommended to use Gaussian filtering before applying gradient operators. Since Gaussian and gradient operations are linear, the Gaussian gradient operator can be defined by taking the derivative of the Gaussian (4.10):

$$h_{\text{Gauss}}(m,n) = m \cdot e^{-\frac{m^2+n^2}{2\sigma^2}}. \tag{4.18}$$

It should be noted that edges are detected when the magnitude of the gradient is maximal. That means the location of edge pixels will not be modified if a mask $\mathbf{h}$ is replaced by $\lambda\mathbf{h}$ with $\lambda \neq 0$. Moreover, the direction of the gradient does not become modified either. For this reason, the elements of $\mathbf{h}$ are usually shown in its simplest way.

An example of the estimation of gradient using the explained masks is illustrated in Fig. 4.16. After the gradient image is calculated, the edges are detected by thresholding. Thus, if the magnitude of the gradient is greater than a certain threshold, then the pixel of the output image is set as an edge pixel. The output for the mentioned example is illustrated in Fig. 4.17. We can see how the boundaries are detected, especially for those objects that are very dark in comparison with their background.

Python Example 4.5: In this example, we show the edge detection of an X-ray image of a pen case using the gradient operators according to the method explained in this Sect. 4.5.1:

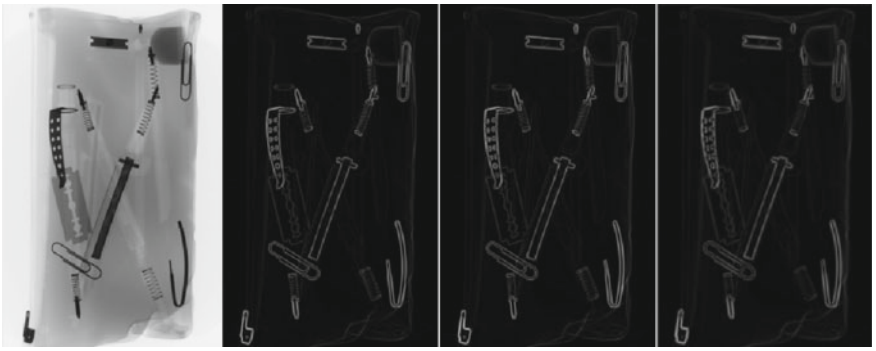


Fig. 4.16 Gradient of an X-ray of a pen case using different masks (Sobel, Prewitt, and Gaussian). See edge detection in Fig. 4.17 [→ Example 4.5]

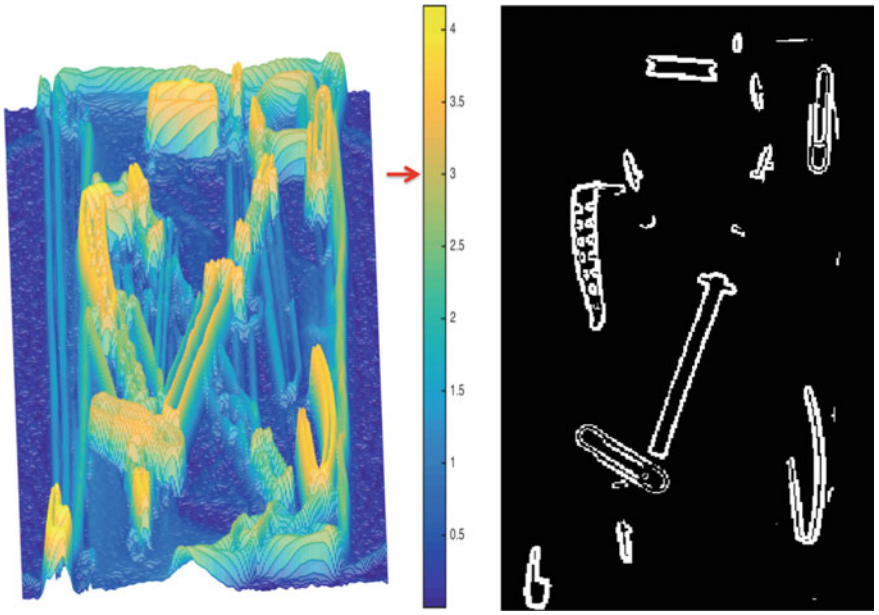


Fig. 4.17 Edge detection by thresholding a Gaussian gradient image of Fig. 4.16. The edges are detected for gradients greater than 3. In this representation, a logarithmical scale for the gray values was used. [→ Example 4.5]

Listing 4.5 : Gradient with different masks

```
import numpy as np
import matplotlib.pylab as plt
import cv2 as cv

from pyxvis.io import gdxraydb
from pyxvis.processing.images import fspecial, linimg, im_grad
from pyxvis.io.visualization import show_image_as_surface


image_set = gdxraydb.Baggages()
img = image_set.load_image(2, 1)
img = cv.resize(img, None, fx=0.25, fy=0.25, interpolation=cv.INTER_AREA)

hs = fspecial('sobel')    # Sobel kernel
hp = fspecial('prewitt')  # Prewitt kernel

hg = fspecial('gaussian', 9, 1.0)
hg = cv.filter2D(hg, cv.CV_64F, np.array([-1, 1]))

gs, __ = im_grad(img, hs)
gp, __ = im_grad(img, hp)
gg, __ = im_grad(img, hg)

gradients = np.hstack([linimg(gs), linimg(gp), linimg(gg)])  # Stack the results as a
    same image.

plt.figure(figsize=(12, 6))
plt.imshow(gradients, cmap='gray')
plt.show()

img_y = np.log(gg + 1)

show_image_as_surface(img_y[-5:5:-1, -5:5:-1], elev=80, azim=-185, fsize=(10, 10),
    colorbar=True)

fig, ax = plt.subplots(1, 1, figsize=(10, 10))
ax.imshow(img_y > 3, cmap='gray')
ax.axis('off')
plt.show()
```

The output of this code is shown in Figs. 4.16 and 4.17. The code uses command im_grad of pyxvis Library. □

4.4.2 Laplacian-of-Gaussian (LoG)

In the previous section, we learned that the edges of a function can be located by detecting local maxima of the magnitudes of gradients. We know that the location of the maximal values of the gradient coincides with zero-crossing of the second derivative. In order to eliminate noisy zero-crossings, which do not correspond to high gradient values, this method uses a Gaussian low-pass filter (see Fig. 4.18). The method, known as Laplacian-of-Gaussian (LoG), is based on a kernel and a zero-crossing algorithm [4]. LoG-kernel involves a Gaussian low-pass filter (4.10), which is suitable for the pre-smoothing of the noisy X-ray images. LoG-kernel is defined as the Laplacian of a 2D-Gaussian function:

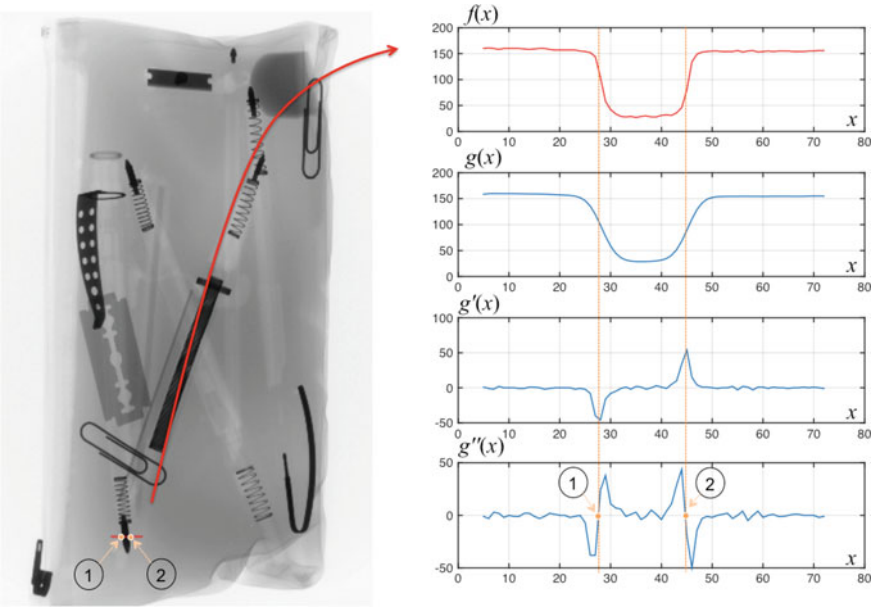


Fig. 4.18 Example of edge detection in 1D using LoG: The profile of the red line in an X-ray image is shown as $f(x)$. This function is filtered by a Gaussian low-pass filter obtaining $g(x)$. The gradient of $g(x)$, represented as $g'(x)$ shows the location of the maximal value (see dashed orange lines), that corresponds to the zero-crossing of the second derivative of $g(x)$. The edges '1' and '2' are then detected

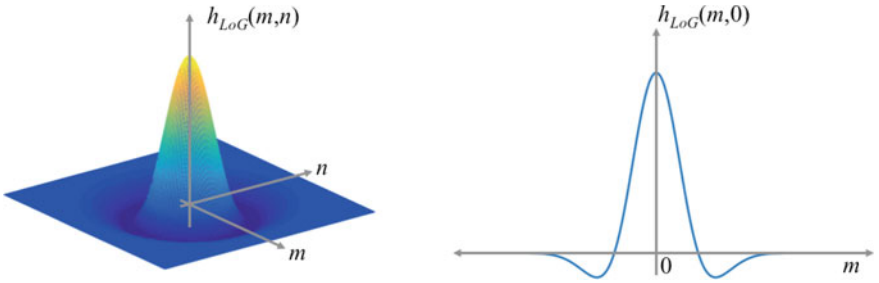


Fig. 4.19 LoG mask: (Left) representation of (4.19), (Right) profile for $n = 0$

$$h_{LoG}(m,n) = \frac{1}{2\pi\sigma^4} \cdot \left(2 - \frac{m^2+n^2}{\sigma^2}\right) \cdot e^{-\frac{m^2+n^2}{2\sigma^2}}. \tag{4.19}$$

LoG-kernel is shown in Fig. 4.19. The parameter σ defines the width of the Gaussian function and, thus, the amount of smoothing and the edges detected (see Fig. 4.20). Using (4.8), we can calculate an image $\mathbf{Y}$ in which the edges of the original image are located by their zero-crossing. After zero-crossing, the detected edges $\mathbf{Z}$

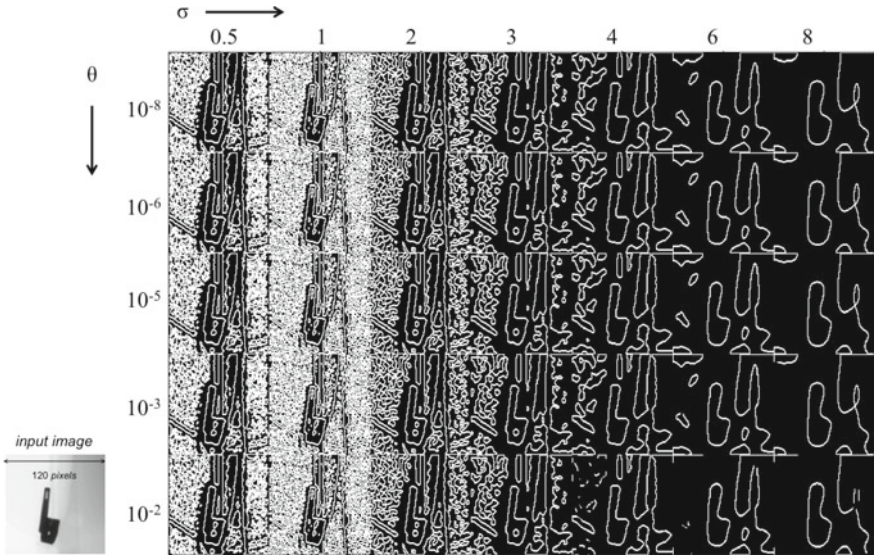


Fig. 4.20 Example of LoG edge detection of a slider (see bottom left of X-ray image of the pen case Fig. 4.18). Several values for σ and θ are presented. The smoothness of the edges is controlled by increasing σ. The reduction of noisy edges is controlled by increasing θ. [→ Example 4.6]

correspond to the maximal (or minimal) values of the gradient image. In order to eliminate weak edges, a threshold θ is typically used. Thus, all edge pixels in $\mathbf{Z}$ that are not strong enough are ignored. The higher the threshold, the less edges will be detected. On the other hand, if $\theta = 0$, i.e., all zero-crossings are included, the edge image has closed and connected contours. As we will see in Sect. 4.5.2, this property is required when segmenting a region of the image.

Python Example 4.6: In this example, we show the edge detection of the object of a pen case (see Fig. 4.20) according to LoG algorithm explained in this Sect. 4.4.2:

Listing 4.6 : Edge detection using LoG

```
import numpy as np
import matplotlib.pylab as plt
import cv2 as cv

from pyxvis.io import gdxraydb
from pyxvis.processing.images import Edge

image_set = gdxraydb.Baggages()

img = image_set.load_image(2, 1)
img = cv.resize(img, None, fx=0.5, fy=0.5, interpolation=cv.INTER_AREA)
img = img[595:715, 0:120]

plt.figure(figsize=(12, 6))
```

```
plt.imshow(img, cmap='gray')
plt.axis('off')
plt.show()

threshold = np.array([1e-8, 1e-6, 1e-5, 1e-3, 1e-2])  # Different threshold values
sigma = np.array([0.5, 1.0, 2.0, 3.0, 4.0, 6.0, 8.0]) # Different sigma values

rows = np.array([])
for t in threshold:
    cols = np.array([])
    for s in sigma:
        detector = Edge('log', t, s)
        detector.fit(img)
        cols = np.hstack([cols, detector.edges]) if cols.size else detector.edges
    rows = np.vstack([rows, cols]) if rows.size else cols

fig, ax = plt.subplots(1, 1, figsize=(12, 8))
ax.imshow(rows, cmap='gray');

# Figure configuration
from matplotlib.ticker import FixedLocator, FixedFormatter
ax.set_title('Sigma', y=1.05)
ax.set_ylabel('Threshold')
ax.tick_params(bottom=False, top=True, left=False, right=True)
ax.tick_params(labelbottom=False, labeltop=True, labelleft=True, labelright=False)
x_formatter = FixedFormatter(sigma)
y_formatter = FixedFormatter(threshold)
x_locator = FixedLocator(60 + 120 * np.arange(sigma.shape[0]))
y_locator = FixedLocator(60 + 120 * np.arange(threshold.shape[0]))
ax.xaxis.set_major_formatter(x_formatter)
ax.yaxis.set_major_formatter(y_formatter)
ax.xaxis.set_major_locator(x_locator)
ax.yaxis.set_major_locator(y_locator)
plt.show()
```

The output of this code—step by step—in Fig. 4.20. The code uses command Edge of pyxvis Library. □

4.4.3 Canny Edge Detector

Canny proposes a 2D linear mask for edge detection based on an optimization approach [2], in which the following criteria are met:

- Good detection: The detection should respond to an edge (and not to noise).
- Good localization: The detected edge should be near the true edge.
- Single response: It should be one detected edge per true edge.

The optimal mask is similar to a derivative of a Gaussian. Thus, the idea is to use this mask to find the local maxima of the gradient of the image. The practical implementation uses adaptive thresholding of the gradient (to detect strong and weak edges) with hysteresis (weak edges are detected only if they are connected to strong edges).

In Example 4.6, the code line `detector = Edge('log',t,s)` can be changed by `E = Edge('canny',t,s)` to elucidate similarities and differences between both edge detectors.

4.5 Segmentation

Image segmentation is defined as the process of subdividing an image into disjointed regions [3]. A region is defined as a set of connected pixels that correspond to a certain *object of interest*. Obviously, these regions of interest depend on the application. For instance, in the inspection of aluminum castings with X-ray images, the idea of segmentation is to find regions with defects. Here, the object of interest is the defects. An example is shown in Fig. 4.21, where the segmentation is the small spots that indicate defective areas.

Another example of segmentation in X-ray testing is weld inspection as illustrated in Fig. 4.22, where a weld seam with two regions is clearly distinguishable: the weld (*foreground*) and the base metal (*background*). In this example, the (first) object of interest is the weld because it is the region where defects can be present. The reader can clearly identify the defects in the weld (see small dark regions in the middle of the X-ray image). In this example, the defects, that have to be detected in a second segmentation stage, are our second object of interest. In this case, the background is the weld, and the foreground is the defects.

Segmentation is one of the most difficult processes in image processing. Clearly, there are some simple applications in which certain segmentation techniques are very effective (e.g., separation between weld and metal base as shown in Fig. 4.22), however, in many other applications segmentation is far from being solved as the appearance of the object of interest can become very intricate. This is the case of

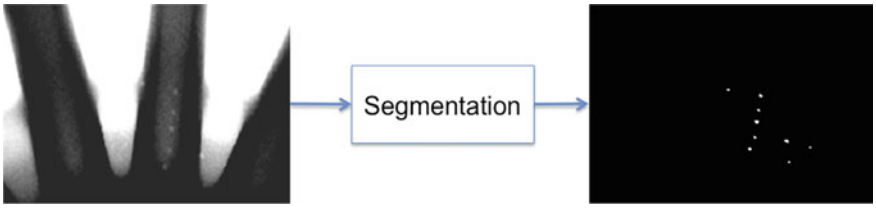


Fig. 4.21 Example of segmentation: detection of defects in an aluminum wheel (see details in Fig. 4.14). [→ Example 4.4]

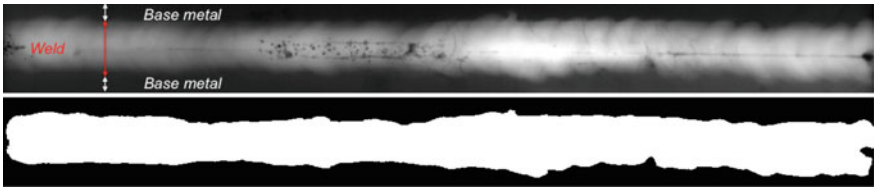


Fig. 4.22 Segmentation of a weld. (Top) original X-ray image. (Bottom) segmentation. The first step in weld inspection is the segmentation of the weld, i.e., the region where the defects can be present (see segmentation in Fig. 4.27). The second step is the detection of defects. [→ Example 4.8]

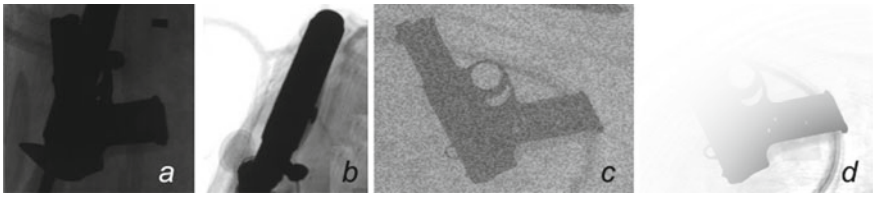


Fig. 4.23 Problems when detecting a gun. Detection can be a very complex task due to **a** occlusion, **b** self-occlusion, **c** noise, **d** wrong acquisition

baggage screening, where the segmentation of objects of interest inside a piece of luggage can be extremely difficult due to problems of (self-)occlusion, noise, and acquisition (see Fig. 4.23).

In image processing for X-ray testing, segmentation is used to detect (potential) regions that can be the objects of interest that we are looking for. As mentioned in previous examples, segmentation divides the X-ray image into two areas: foreground and background. Foreground means the pixels of the object(s) of interest. Background means the remaining pixels of the image. Usually, a *binary image* is the output of the segmentation process as we can see in Figs. 4.21 and 4.22, where a pixel equals to '1' (white) is foreground, whereas '0' (black) means background. We use the term 'potential' throughout to make it clear that a segmented region is not necessarily the final detected region. In many applications, the segmentation is just the first step of the whole detection process. In such cases, an additional step that analyzes the segmented region is required. This additional step can include multiple view analysis or a pattern recognition technique (see Fig. 1.22). The later extracts and classifies features of the segmented region in order to verify whether it corresponds to the object that we are detecting or it is a *false detection*.

Thus, segmentation basically acts as a focus of attention mechanism that filters the information that is fed to the following steps, as such a failure in the segmentation is catastrophic for the final performance. In this section, we will review some basic segmentation techniques that have been used in X-ray testing: thresholding (Sect. 4.5.1), region growing (Sect. 4.5.2), and maximally stable extremal regions (Sect. 4.5.3). Please note that more complex techniques based on computer vision algorithms will be addressed in the next sections.

4.5.1 Thresholding

In some X-ray images, we can observe that the background is significantly darker than the foreground. This is the case of an X-ray image of an apple placed on a uniform background as illustrated in Fig. 4.24. It is clear that the object of interest can be segmented using a very simple approach based on *thresholding*. In this section, we will explain a methodology based on two steps: (i) estimate of a global

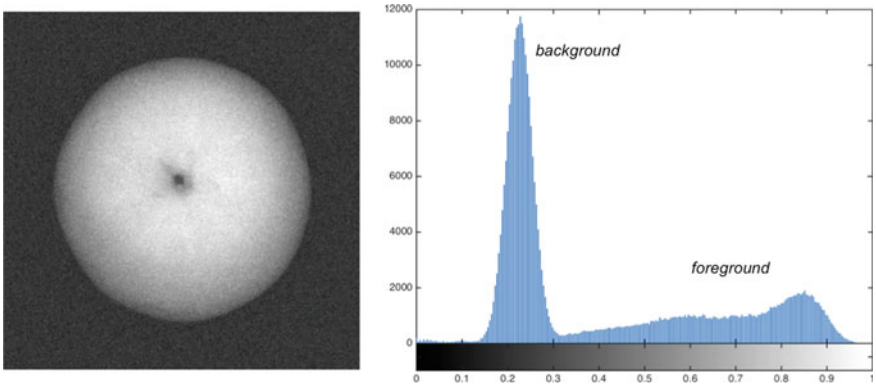


Fig. 4.24 X-ray image of an apple and its histogram

threshold using a statistical approach and (ii) a morphological operation in order to fill the possible holes presented in the segmented binary image. This method was originally presented for color food images [14], however, it can be easily adapted for X-ray images.

The X-ray image to be segmented is stored in matrix **I**. In order to enhance the contrast of the image, a linear transformation can be performed (see Sect. 4.2.2). Additionally, a linear or non-linear filter can be used for noise removal (see Sect. 4.3). Here, after image enhancement and filtering, we obtain a new image **J**, where $J_{\max} = 1$ and $J_{\max} = 0$. Image **J** has a bimodal histogram as shown in Fig. 4.24, where the left distribution corresponds to the background and the right to the food image. In this image, the first separation between foreground and background can be performed estimating a global threshold t. Thus, we define a binary image

$$K(i,j) = \begin{cases} 1 & \text{if} \quad J(i,j) > t \\ 0 & \text{else} \end{cases} \tag{4.20}$$

where '1' means foreground and '0' background, that define two classes of pixels in the image. Figure 4.25 illustrates different outputs depending on t. The problem is to determine the 'best' threshold t that separates the two modes of the histogram from each other. A good separation of the classes is obtained by ensuring (i) a small variation of the gray values in each class, and (ii) a large variation of the gray values in the image [6]. The first criterion is obtained by minimizing a weighted sum of the within-class variances (called *intraclass* variance $\sigma_W^2(t)$):

$$\sigma_W^2(t) = p_b(t)\sigma_b^2(t) + p_f(t)\sigma_f^2(t), \tag{4.21}$$

where the indices 'b' and 'f' denote respectively background and foreground classes, and p and σ^2 are respectively the probability and the variance for the indicated class. These values can be computed from the histogram.

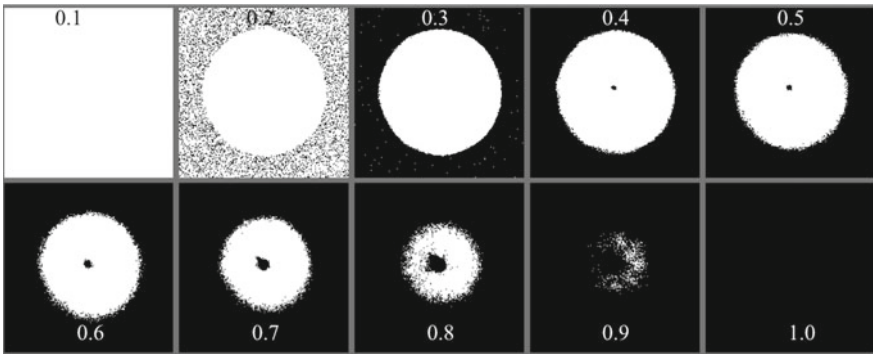


Fig. 4.25 Segmentation using threshold $t = 0.1, 0.2, \ldots 1.0$

The second criterion is obtained by maximizing the between-class variance (called *interclass* variance $\sigma_B^2(t)$):

$$\sigma_B^2(t) = p_b(\mu_b(t) - \mu)^2 + p_f(\mu_f(t) - \mu)^2, \tag{4.22}$$

where μ_b, μ_f, and μ indicate the mean value of the background, foreground, and the whole image respectively.

The best threshold t can be estimated by a sequential search through all possible values of t that minimizes $\sigma_W^2(t)$ (or maximizes $\sigma_B^2(t)$). Both criteria, however, lead to the same result because the sum $\sigma_W^2 + \sigma_B^2$ is a constant and corresponds to the variance of the whole image [6]. Skimage Library computes the global image threshold by minimizing the intraclass variance $\sigma_W^2(t)$. The threshold can be obtained with the function `threshold_otsu`. In our example, the obtained threshold is $t = 0.4824$, that is approximately 0.5 (see Fig. 4.25).

We can observe in Fig. 4.25 that the segmentation suffers from inaccuracy because there are many dark (bright) regions belonging to the foreground (background) that are below (above) the chosen threshold and therefore misclassified. For this reason, additional morphological processing must be obtained.

The morphological operation is performed in three steps as shown in Fig. 4.26: (i) remove small objects, (ii) close the binary image, and (iii) fill the holes.

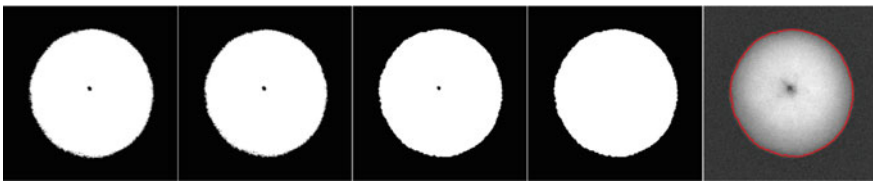


Fig. 4.26 Additional morphological operations. From left to right: **K**: binary image after thresholding, **A**: after removal of small objects, **C**: after closing process, **R**: after filling holes, and boundary superimposed onto the original image. [→ Example 4.7 ]

In the first step, we remove from binary image **K** obtained from (4.20) all connected regions that have fewer than n pixels (see image **A** in Fig. 4.26).This operation is necessary to eliminate those isolated pixels of the background that have a gray value greater than the selected threshold. Empirically, we set $n = NM/100$, where $N \times M$ is the number of pixels of the image.

The second step *closes* the image, i.e., the image is *dilated* and then *eroded*. The dilation is the process that incorporates into the foreground the background pixels that touch it. On the other hand, erosion is the process that eliminates all the boundary pixels of the foreground. The closing process (dilation followed by erosion) fills small holes and thins holes in the foreground, connecting nearby regions, and smoothing the boundaries of the foreground without changing the area significantly [3] (see image **C** in Fig. 4.26). This operation is very useful in objects that have spots in the boundary.

Finally, the last operation fills the holes in the closed image (see image **R** in Fig. 4.26). We use this operation to incorporate into the foreground all pixels '0' that are inside of the region. The whole algorithm is implemented in command seg_bimodal of pyxvis Library. In the implementation, as suggested in [14], an offset p that modifies the threshold is used because there are dark zones in the boundary that are not well included in the original segmented region.

 Python Example 4.7: In this example, we show the segmentation of an X-ray image of an apple (see Fig. 4.24) according to the method explained in this Sect. 4.5.1:

Listing 4.7 : Apple segmentation using global thresholding

```
import matplotlib.pylab as plt

from pyxvis.io import gdxraydb
from pyxvis.processing.segmentation import seg_bimodal
from pyxvis.io.visualization import binview


image_set = gdxraydb.Nature()
img = image_set.load_image(5, 9)

mask, contours = seg_bimodal(img)
seg = binview(img, mask, 'g')

fig, ax = plt.subplots(1, 2, figsize=(12, 6))
ax[0].imshow(img, cmap='gray')
for n, contour in enumerate(contours):
        ax[0].plot(contour[:, 1], contour[:, 0], color='r', linewidth=3)
ax[0].axis('off')
ax[1].imshow(seg)
ax[1].axis('off')
plt.show()
```

The output of this code—step by step—in Fig. 4.26. The code uses command seg_bimodal of pyxvis Library. □

The above-mentioned methodology, based on a *global threshold*, does not segment appropriately when there is a large variation in the background or foreground intensity. For this reason, in certain cases, it is recommended to use an *adaptive threshold*. The idea is to divide the input image into partitions with some overlapping. Each partition is handled as a new image that is segmented by thresholding (using a global but an ad hoc threshold for each partition). The output image is a fusion of all segmented partitions, e.g., using logical OR operator. The next example shows an implementation that was used to segment the weld of Fig. 4.22. Since the weld area is horizontal, the proposed method uses vertical partitions that include background and foreground areas. The segmentation of each partition is performed by the same method used for the segmentation of the apple.

 Python Example 4.8: This example shows the segmentation of a weld of Fig. 4.22 using adaptive thresholding. The approach is simple; the input image is divided into four partitions with an overlapping of 50%. Each partition is segmented using command seg_bimodal of pyxvis Library. The obtained binary images of the segmentation are superimposed using logical OR operator:

Listing 4.8 : Weld segmentation using adaptive thresholding

```
import numpy as np
import matplotlib.pylab as plt

from skimage.measure import find_contours

from pyxvis.io import gdxraydb
from pyxvis.processing.segmentation import seg_bimodal
from pyxvis.io.visualization import binview


image_set = gdxraydb.Welds()
img = image_set.load_image(1, 1)

mask = np.zeros(img.shape, np.uint8)  # Create a uint8 mask image
max_width = img.shape[1]

d1 = int(np.round(max_width/4))
d2 = int(np.round(d1 * 1.5))

i1 = 0

while i1 < max_width:
    i2 = min(i1 + d2, max_width)  # second column of partition
    img_i = img[:, i1:i2]  # partition i
    bw_i, _ = seg_bimodal(img_i)  # segmentation of partition i
    roi = mask[:, i1:i2]
    overlap = np.bitwise_or(roi, bw_i)  # addition into whole segmentation
    mask[:, i1:i2] = overlap
    i1 = i1 + d1  # update of first column

seg = binview(img, mask, 'g', g=5)

contours = find_contours(np.float32(mask), 0.5)

fig, ax = plt.subplots(2, 1, figsize=(14, 5))
ax[0].imshow(img, cmap='gray');
for n, contour in enumerate(contours):
        ax[0].plot(contour[:, 1], contour[:, 0], color='r', linewidth=3)
```

```
ax[0].axis('off')
ax[1].imshow(seg)
ax[1].axis('off')
fig.tight_layout()
plt.show()
```

The output of this code—step by step—is shown in the last row of Fig. 4.27. □

4.5.2 Region Growing

In region growing, we segment a region using an iterative approach. We start by choosing a seed pixel, as shown in Fig. 4.28. At this moment, our region is initialized and its size is one pixel only. We extract some feature of the region, e.g., the gray value. We extract the same feature of each neighboring pixel. In our example, there are four neighbors (up, down, right, and left), as we can see in third image of Fig. 4.28. We increase our region by adding similar neighboring pixels, i.e., those neighboring pixels that have a similar feature to the region. The whole process is continued, each added pixel is a new seed for the next iteration until no more neighboring pixels can be added.

In Fig. 4.28, we have a binary edge image. The feature that we use to establish the similarity is the value of the pixel. In our example, there are only two pixel values: '0' for the edge pixels, and '1' for the remaining pixels. That means that the value of the pixel of the seed is '1' and in each iteration, we can add only those neighboring pixels the value of which are '1'. As we can see, the red region grows up from 1

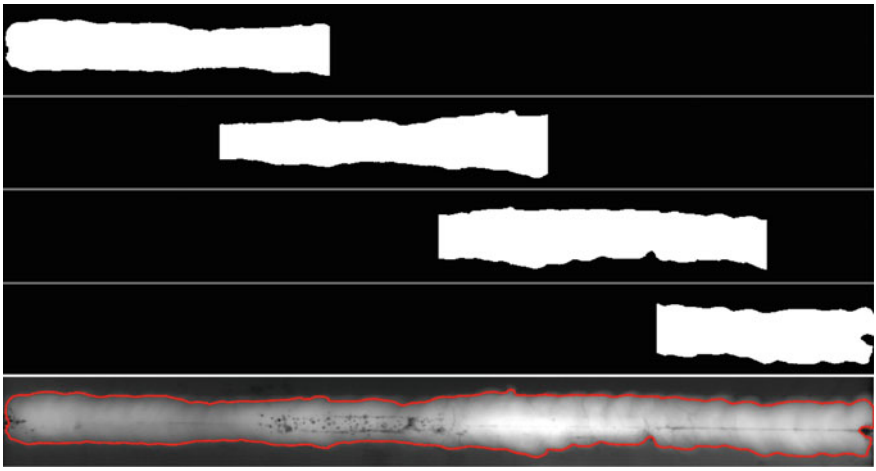


Fig. 4.27 Weld segmentation of Fig. 4.22 using adaptive thresholding of four partitions. The last image shows the segmentation after fusion the four individual segmentations using logical OR operator.) [→ Example 4.8]

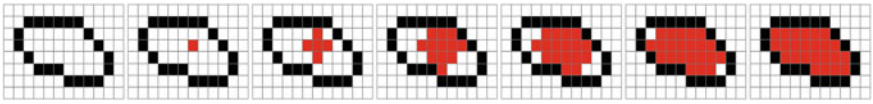


Fig. 4.28 Region growing: we start with a seed pixel that grows in each iteration in four directions until a boundary is found. The directions in this example are four: up, down, right, and left

Fig. 4.29 Region growing in an X-ray image using a seed pixel in the object of interest. The region is well segmented as we can see in the binary image and in boundaries. [→ Example 4.9]

pixel to 5, 12, 16, 22, and finally, 24 pixels. The output is the red region of the last step.

Region growing can be used directly in X-ray images as illustrated in Fig. 4.29. We start with a seed pixel, and neighboring pixels are added if they are similar enough.

Python Example 4.9: In this example, we show the performance of region growing in the segmentation of an object in an X-ray image of a pen case (see Fig. 4.29). The seed is chosen at pixel (190,403). The seed grows by adding neighboring pixels with similar gray values. We use command region_growing of pyxvis Library. In this implementation, the similarity between region and neighboring pixels is established if $|\bar{R} - r_n| \leq \theta$, where $\bar{R}$ is the average of the gray values of the region, r_n is the gray value of the neighboring pixel, and θ is a threshold. In this example, $\theta = 20$:

Listing 4.9 : Region Growing

```
import matplotlib.pylab as plt
import cv2 as cv

from pyxvis.io import gdxraydb
from pyxvis.processing.segmentation import region_growing
from pyxvis.io.visualization import binview
```

```
image_set = gdxraydb.Baggages()

img = image_set.load_image(3, 4)
img = cv.resize(img, None, fx=0.35, fy=0.35, interpolation=cv.INTER_AREA)

th = 40  # threshold
si, sj = (403, 190)  # Seed

mask = region_growing(img, (si, sj), tolerance=th)

seg = binview(img, mask, 'g')

fig, ax = plt.subplots(1, 3, figsize=(14, 8))
ax[0].imshow(img, cmap='gray')
ax[0].plot(sj, si, 'r+')
ax[0].axis('off')
ax[1].imshow(mask, cmap='gray')
ax[1].axis('off')
ax[2].imshow(seg)
ax[2].axis('off')
plt.tight_layout()
plt.show()
```

The output of this code is shown in Fig. 4.29. □

Region growing can be used in X-ray testing in defect detection (see, for example, interesting approaches in aluminum castings [10] and welds [11]). The method is illustrated in Fig. 4.30. The method uses an edge detection algorithm to obtain an edge image with closed and connected contours around the real defects. Thus, we use region growing to isolate each region enclosed by edges. The idea is to extract features from this isolated region (e.g., area, average of gray value, contrast, etc.) that can be used in a classification strategy. In our example, a region is segmented using a very simple classifier (the features of a segmented region must be in certain ranges, e.g., $A_{\min} \leq \text{Area} \leq A_{\max}$). Obviously, more sophisticated features and classifiers can be used to improve the segmentation performance in more complex scenarios as we will see in the following chapters.

Python Example 4.10: In this example, we show how to segment defects in aluminum castings using binary images of potential defects and some simple features that can be extracted from each potential region. In this example, we segment all those regions the area of which is between 200 and 2000 pixels, the average of the gray value is less than 150, and the contrast is greater than 1.1:

Listing 4.10 : Detection of defects in castings

```
import numpy as np
import matplotlib.pylab as plt

from skimage.segmentation import find_boundaries

from pyxvis.io import gdxraydb
from pyxvis.processing.segmentation import seg_log_feature
from pyxvis.io.visualization import binview


image_set = gdxraydb.Castings()
X = image_set.load_image(31, 19)
```

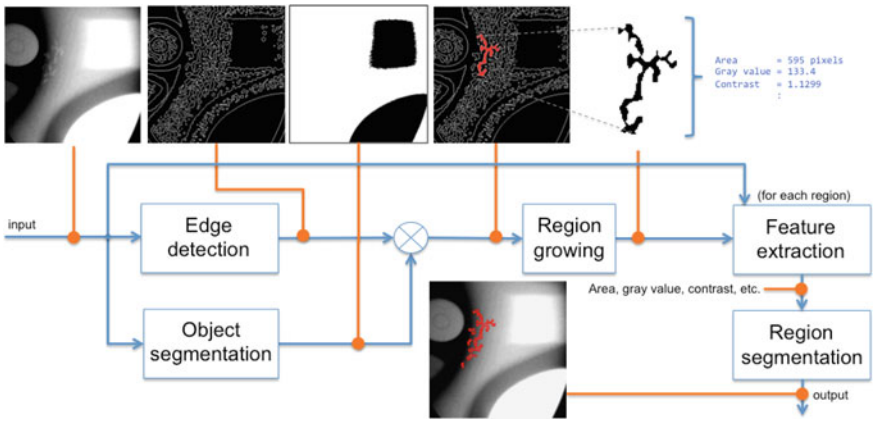


Fig. 4.30 Segmentation of defects in aluminum castings using region growing, edge detection, and some features. The size of the image in this example is 286 × 286 pixels [→ Example 4.10]

```
X = X[0:572:2, 0:572:2]  # Donwsampling the image

fig1, ax1 = plt.subplots(1, 1, figsize=(8, 8))
ax1.imshow(X, cmap='gray')
ax1.set_title('Input image')
ax1.axis('off')
plt.show()

R = X < 240

fig2, ax2 = plt.subplots(1, 1, figsize=(8,8))
ax2.imshow(R, cmap='gray')
ax2.set_title('Segmented object')
ax2.axis('off')
plt.show()

options = {
    'area': (30, 1500),  # Area range (area_min, area_max)
    'gray': (0, 150),  # Gray value range (gray_min, gray_max)
    'contrast': (1.08, 1.8),  # Contras range (cont_min, cont_max)
    'sigma': 2.5
}

Y, m = seg_log_feature(X, R, **options)

print(f'Found {m} regions.')

fig3, ax3 = plt.subplots(1, 1, figsize=(8, 8))
ax3.imshow(binview(X, find_boundaries(Y)), cmap='gray')
ax3.set_title('Segmented regions')
ax3.axis('off')
plt.show()
```

.

The output of this code is shown—step by step—in Fig. 4.30. We use command seg_log_feature of pyxvis Library. □

This method is very effective for regions of interest that have gray values significantly different from the background (the reader, for instance, can try to segment

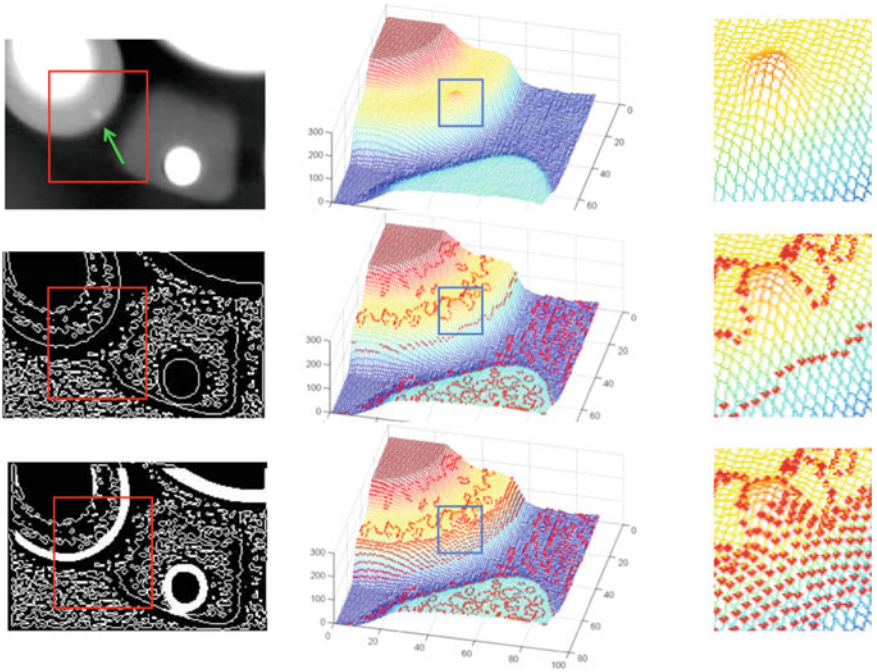


Fig. 4.31 X-ray image of an aluminum casting with a small defect at an edge (see defect pointed by green arrow). (First row) original image. (Second row) LoG. (Third row) LoG and high gradient pixels. (First column) image representation. (Second column) 3D representation of red square. (Third column) zoom of blue square. In this representation, the edge pixels are represented as red points superimposed onto the 3D surface. The output of this method is a binary image in which the real defects are closed by edges

the objects of the pen case of Fig. 4.29 using command seg_log_feature of pyxvis Library).

Nevertheless, the method may fail if the boundaries do not close a region of interest. This is the case in some defects of aluminum castings that are at an edge of a regular structure as illustrated in Fig. 4.31.[2] In this problem, we can see that the edges of LoG algorithm (and other edge detection algorithms like Sobel or Canny as well) cannot correctly find the defect's edge. Contrarily, it finds the regular structure's edge. To overcome this problem, we have to complete the remaining edges of these defects. A simple approach was suggested in [13] by thickening of the edges of the regular structure after LoG-edge detection: (i) The gradient of the original image is calculated. The gradient image is computed by taking the square root of the sum of the squares of the gradient in horizontal and in vertical directions. These are calculated by the convolution of the radioscopic image with the first derivative (in the corresponding direction) of the Gaussian low-pass filter used in the LoG fil-

[2] A video of this small defect can be watched at http://youtu.be/e3wDJhq2Tqg.

ter. (ii) High gradient pixels are detected by thresholding. (iii) The resulting image is added to the LoG-edge detection image. Afterwards, each closed region is segmented as a potential flaw. As can be observed the effectiveness of this method in Fig. 4.31, the defect on an edge of a regular structure could be satisfactorily closed. Thus, the method of Fig. 4.30 can be used.

4.5.3 Maximally Stable Extremal Regions (MSER)

In order to understand the MSER approach [8], the reader can imagine a simple video as follows. The video will have 256 frames. Frame t is defined as the binary image $\mathbf{I} < t$, where $\mathbf{I}$ is the input image we want to segment. If the binary image is black for '0' and white for '1', at the beginning our video will be very dark and at the end very bright. In the middle, we will have some regions depending on the threshold.[3] Thus, each region has an area $A(t)$, that depends on t. If the gray value of the region is very different from its background, the area of this region will be stable for some thresholds $t, t+1 \ldots t+p$, i.e., $A(t) \approx A(t+1) \cdots \approx A(t+p)$. The key idea of MSER is to segment those regions which fulfill:

$$\frac{\Delta A}{\Delta t} < \theta, \tag{4.23}$$

where θ is a threshold. That means those regions whose sizes remain approximately stable by varying the segmentation threshold t are to be detected.

Python Example 4.11: In this example, we show the segmentation of an X-ray image of a pen case (see Fig. 4.15) according to MSER approach (see Sect. 4.5.3):

Listing 4.11 : Pencase segmentation using MSER algorithm

```
import numpy as np
import matplotlib.pylab as plt
import cv2 as cv

from skimage.segmentation import find_boundaries
from skimage.morphology import binary_dilation

from pyxvis.io import gdxraydb
from pyxvis.processing.segmentation import seg_mser
from pyxvis.io.visualization import plot_bboxes


image_set = gdxraydb.Baggages()
img = image_set.load_image(2, 1)

fig1, ax1 = plt.subplots(1, 1, figsize=(10, 10))
ax1.imshow(img, cmap='gray')
ax1.set_title('Input image')
```

[3] The video can be found in http://youtu.be/tWdJ-NFE6vY.

```
ax1.axis('off')
plt.show()

mser_options = {
    'area': (60, 40000),  # Area of the ellipse (Max, Min)
    'min_div': 0.9,  # Minimal diversity
    'max_var': 0.2,  # Maximal variation
    'delta': 3,  # Delta
}

J, L, bboxes = seg_mser(img, **mser_options)

E = binary_dilation(find_boundaries(J, connectivity=1, mode='inner'), np.ones((3, 3)))

fig2, ax2 = plt.subplots(1, 1, figsize=(10, 10))
ax2.imshow(E, cmap='gray')
ax2.set_title('Edges')
ax2.axis('off')
plt.show()

fig3, ax3 = plt.subplots(1, 1, figsize=(10, 10))
ax3.imshow(L, cmap='gray')
ax3.set_title('Segmentation')
ax3.axis('off')
plt.show()

fig4, ax4 = plt.subplots(1, 1, figsize=(10, 10))
ax4.imshow(img, cmap='gray')
ax4 = plot_bboxes(bboxes, ax=ax4)
ax4.set_title('Bounding Boxes')
ax4.axis('off')
plt.show()
```

The output of this code—step by step—in Figs. 4.15 and 4.32. The code uses command seg_mser of pyxvis Library. This function uses OpenCV implementation of MSER. □

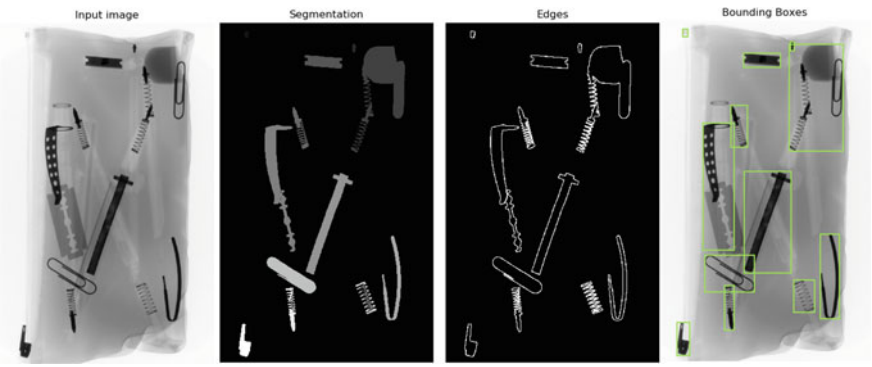


Fig. 4.32 Segmentation of objects in a pencase using MSER [→ Example 4.11]

4.6 Image Restoration

Image restoration involves recovering detail in severely blurred images. This process is more efficient when the causes of the imperfections are known a priori [5]. This knowledge may exist as an analytical model, or as a priori information in conjunction with knowledge (or assumptions) of the physical system that provided the imaging process in the first place. The purpose of restoration then is to estimate the best source image, given the blurred example and some a priori knowledge.

In this section, we concentrate on the particular case of blur caused by uniform linear motion, which may be introduced by relative motion between detector and object. Early work on restoring an image degraded by blurring calculated the deblurring function as an inverse filtering. The inverse filtering evaluation of the blurring function h (or point spread function PSF) in the frequency domain tends to be very sensitive to noise [5]. The cause of this sensitivity is the lowpass nature of the PSF: its frequency response $H(\omega)$ contains very small values, and small noise in the frequency regions where $1/H(\omega)$ is very large, maybe greatly emphasized. Sondhi [5] proposed a non-iterative algorithm to find a solution to the uniform-blurring case, but the computational load is extremely high in small motions. Another two non-iterative approaches are presented in [9]. In the first one, the matrix left division calculates the restored signal as a signal that has the fewest possible nonzero components. This solution differs strongly from the original signal because the original signal must not have necessarily many zero components. The second one, the Moore–Penrose pseudo-inverse of a matrix, finds a restored signal whose norm is smaller than any other solution. This solution is very good, but the estimation is based on Singular Value Decomposition (SVD), whose computation load is very high. In this section, we address the above problems and reduce the computational times significantly using a new technique that minimizes the norm between blurred and original.

A blurred X-ray image $g(x, y)$ that has been degraded by a motion in the vertical direction x and the horizontal direction y can be modeled by

$$g(x, y) = \frac{1}{T} \int_0^T f(x - x_t(t), y - y_t(t))dt, \tag{4.24}$$

where f, T, $x_t(t)$ and $y_t(t)$ represent respectively the deterministic original X-ray image, the duration of the exposure and the time-varying component of motion in the x and y directions. In this case, the total exposure is obtained by integrating the instantaneous exposure over the time interval during which the shutter is open. By rotation of the camera or by using a transformation that rotates the blurred image, a new system of coordinates is chosen in which $x_t(t)$ is zero. Considering that the original image $f(x, y)$ undergoes uniform linear motion in the horizontal direction y only, at a rate given by $y_t(t) = ct/T$, let us write (4.24), with $u = y - ct/T$, as follows:

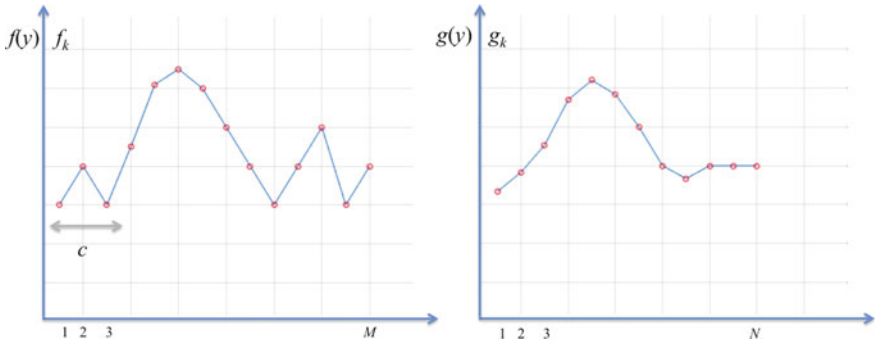


Fig. 4.33 Blurring process: (Left) original row f. (Right) Blurred row

$$g(y) = \frac{1}{T}\int_0^T f(y - ct/T)dt = \frac{1}{c}\int_{y-c}^{y} f(u)dt, \tag{4.25}$$

or as a digital that has been discretized in spatial coordinates by taking N samples $\Delta y = Y/N$ units apart:

$$g_k = \frac{1}{n}\sum_{i=0}^{n-1} f_{k+i}, \tag{4.26}$$

where

$$g_k = g\left(y_0 + (k-1)\frac{c}{n}\right), \quad f_k = f\left(y_0 + (k-1)\frac{c}{n} - c\right), \tag{4.27}$$

with $n = c/\Delta y$. Figure 4.33 shows a row $\mathbf{f} = [f_1 \ ... \ f_M]^\mathsf{T}$ of an original image and its corresponding row $\mathbf{g} = [g_1 \ ... \ g_N]^\mathsf{T}$ of the blurred image for $n = 3$ pixels. Equation (4.26) describes an underdetermined system of N simultaneous equations (one for each element of vector $\mathbf{g}$) and $M = N + n - 1$ unknowns (one for each element of vector $\mathbf{f}$) with $M > N$. This process is carried out for each row of the image. The degradation of $\mathbf{f}$ can be modeled using a convolution of $\mathbf{f}$ with $\mathbf{h}$, where $\mathbf{h}$ is the PSF, a n-element vector defined as the impulse response of this linear system [5]. Thus, element g_i of vector $\mathbf{g}$ is calculated as a weighted sum of n elements of $\mathbf{f}$, i.e., $g_i = h_1 f_i + h_2 f_{i+1} + ... + h_n f_{i+n-1}$, for $i = 1, ..., N$. Using a circulant matrix, the convolution can be written as $\mathbf{Hf} = \mathbf{g}$:

$$\mathbf{g} = \mathbf{f} * \mathbf{h} = \begin{bmatrix} h_1 & ... & h_n & 0 & 0 & 0 & 0 \\ 0 & h_1 & ... & h_n & 0 & 0 & 0 \\ & & \vdots & & \vdots & & \\ 0 & 0 & ... & 0 & h_1 & ... & h_n \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_M \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_N \end{bmatrix} \tag{4.28}$$

Fig. 4.34 Degradation of an X-ray image of 2208 × 2688 pixels: Original image and degraded images with $n = 32$, 256, and 512 pixels

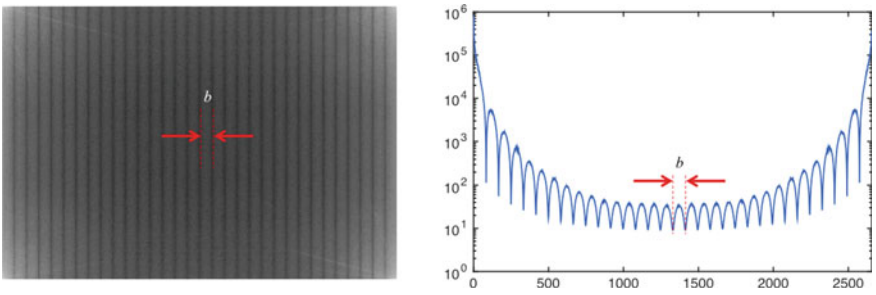


Fig. 4.35 Spectrum of a blurred image which was degraded by uniform linear motion with $n = 32$ pixels. (Left) 2D-Fourier Transformation of the original image of Fig. 4.34. Right Mean of the rows of the Fourier transformation. The size of the degraded image is 2208 × 2657 pixels. It can be demonstrated that b is approximately $2657/n$

An example of a degradation process is shown in Fig. 4.34. In this example, we can see how the objects cannot be recognized when the degradation is severe.

If the PSF is not exactly known, but if we know that it corresponds to a uniform linear motion, the parameter n can be estimated from the spectrum of the blurred image. An example is shown in Fig. 4.35. The 2D-Fourier Transformation of a blurred test is represented in Fig. 4.35a, in this case, a horizontal degradation took place with $n = 32$. The mean of its rows is illustrated in Fig. 4.35b. We can observe that the period of this function is inversely proportional to the length of the blurring process in pixels.

The problem of restoring an X-ray that has been blurred by uniform linear motion consists of solving the underdetermined system (4.28). The objective is to estimate an original row per row ($\mathbf{f}$), given each row of a blurred ($\mathbf{g}$) and a priori knowledge of the degradation phenomenon ($\mathbf{H}$). Since there is an infinite number of exact solutions for $\mathbf{f}$ in the sense that $\mathbf{Hf} - \mathbf{g} = \mathbf{0}$, an additional criterion that finds a sharp restored is required.

We observed that most solutions for $\mathbf{f}$ strongly oscillate. Figure 4.36 shows an example in which four different solutions for $\mathbf{f}$ are estimated, all solutions satisfy Eq. (4.28): $\mathbf{Hf} = \mathbf{g}$. Although these solutions are mathematically right, they do not correspond to the original signal. By the assumption that the components of the higher frequencies of $\mathbf{f}$ are not so significant in the wanted solution, these oscil-

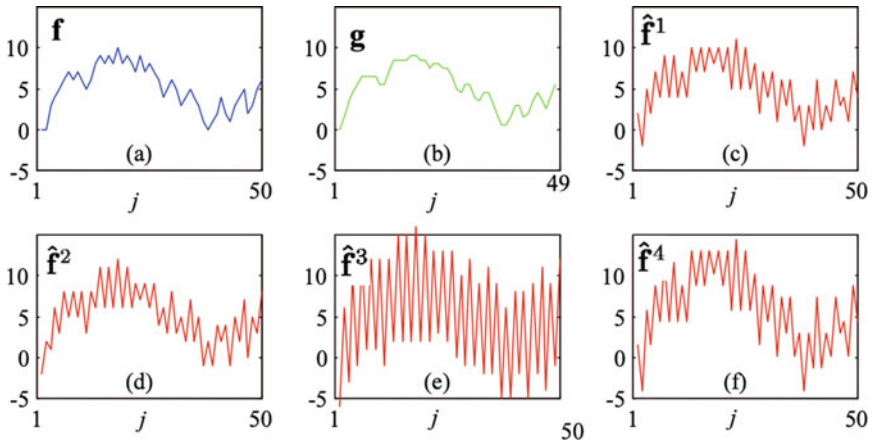


Fig. 4.36 Restoration of row **f**: **a** original row, **b** degraded row with $n = 2$, **c**, **d**, **e** and **f** four possible solution that satisfy $\mathbf{H}\hat{\mathbf{f}} = \mathbf{g}$

lations can be reduced by minimization of the distance between f_k and g_k, i.e., we take a vector as a sharp solution of $\mathbf{Hf} = \mathbf{g}$, so that this presents the smallest distance between original signal and blurred signal: we seek then to minimize the objective function

$$J(\mathbf{f}, \mathbf{g}) = \sum_{k=1}^{N} (f_k - g_k)^2 \rightarrow \min . \tag{4.29}$$

The application of criteria of the minimization of the norm between input and output (MINIO) does not mean that **f** is equal to **g** because this solution does not satisfy the system of equations (4.28) and the size of **f** and **g** are different. The solution also is defined as the vector in the solution space of the underdetermined system $\mathbf{Hf} = \mathbf{g}$ whose first N components has the minimum distance to the measured data, i.e., where the first N elements are of **f**. We can express vector $\hat{\mathbf{f}} = \mathbf{Pf}$, with **f** a $N \times M$ matrix which projects the vector **f** on the support of **g**:

$$\mathbf{P} = \begin{bmatrix} 1\ 0\ \ldots\ 0\ 0\ \ldots\ 0 \\ 0\ 1\ \ldots\ 0\ 0\ \ldots\ 0 \\ \vdots \qquad\qquad \vdots\ 0 \\ 0\ 0\ \ldots\ 1\ 0\ \ldots\ 0 \end{bmatrix} . \tag{4.30}$$

The original optimization problem is now:

$$\hat{\mathbf{f}} = \underset{\mathbf{f}}{\operatorname{argmin}} \parallel \mathbf{Pf} - \mathbf{g} \parallel^2 \tag{4.31}$$

subject to the constraint $\| \mathbf{Pf} - \mathbf{g} \|^2 = \mathbf{0}$. Applying the technique of *Lagrange multipliers* this problem can be alternatively formulated as an optimization problem without constraints:

$$V(\mathbf{f}) = \lambda \| \mathbf{Hf} - \mathbf{g} \|^2 + \| \mathbf{Pf} - \mathbf{g} \|^2 \rightarrow \min, \tag{4.32}$$

if λ is large enough (e.g., $\lambda = 10^6$). The solution of this problem can be easily obtained by computing the partial derivative of criterion V with respect to the unknown $\mathbf{f}$:

$$\frac{\partial}{\partial \mathbf{f}} V(\mathbf{f}) = 2\lambda \mathbf{H}^\mathsf{T}(\mathbf{Hf} - \mathbf{g}) + 2\mathbf{P}^\mathsf{T}(\mathbf{Pf} - \mathbf{g}) = \mathbf{0}, \tag{4.33}$$

then is

$$\hat{\mathbf{f}} = \left[\lambda \mathbf{H}^\mathsf{T}\mathbf{H} + \mathbf{P}^\mathsf{T}\mathbf{P}\right]^{-1} [\lambda \mathbf{H} + \mathbf{P}]^\mathsf{T} \mathbf{g}. \tag{4.34}$$

This solution for the example of Fig. 4.36b is almost identical to the original sharp input signal of Fig. 4.36a. Figure 4.37 shows three different restoration examples.

Python Example 4.12: In this example, we simulate an X-ray image that has been degraded by a horizontal motion. The image is restored using MINIO algorithm (4.34):

Listing 4.12 : X-ray image restoration.

```
import numpy as np
import matplotlib.pylab as plt
import cv2 as cv

from pyxvis.io import gdxraydb
from pyxvis.processing.images import res_minio

image_set = gdxraydb.Baggages()
img = image_set.load_image(46, 90)

n = 128
h = np.ones((1, n)) / n

img_g = cv.filter2D(img.astype('double'), cv.CV_64F, h)
fs = res_minio(img_g, h, method='minio')

fig, ax = plt.subplots(1, 3, figsize=(16, 8))
ax[0].imshow(img, cmap='gray')
ax[0].set_title('Original image')
ax[0].axis('off')
ax[1].imshow(img_g, cmap='gray')
ax[1].set_title('Degraded image')
ax[1].axis('off')
ax[2].imshow(fs, cmap='gray')
ax[2].set_title('Restored image')
ax[2].axis('off')
plt.show()
```

The output of this code is shown in the last row of Fig. 4.37. Details of the baggage are not discernible in the degraded image but are recovered in the restored image.

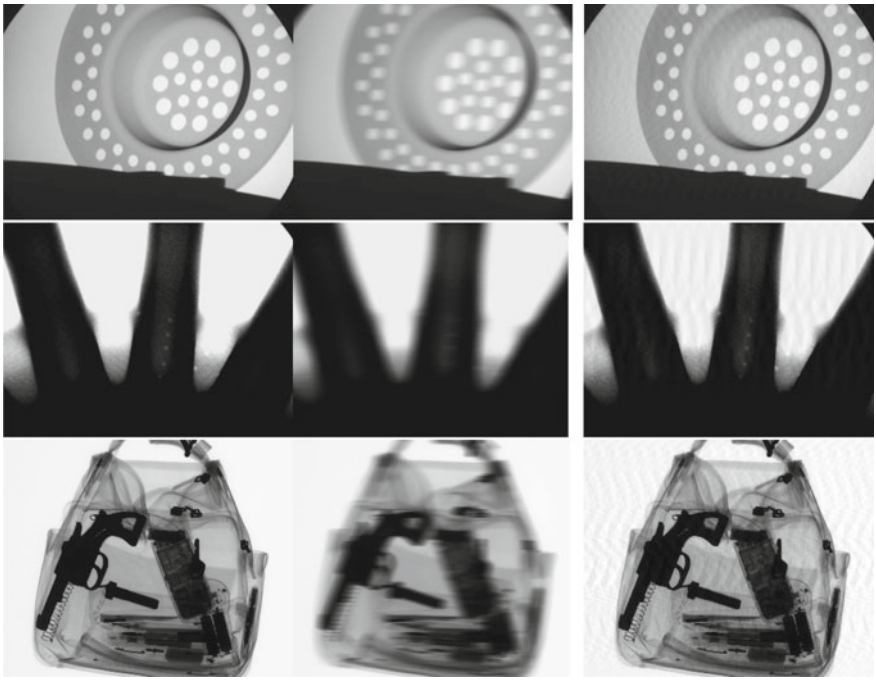


Fig. 4.37 Restoration in simulated degraded X-ray images. Each column shows the original, the degraded with n pixels, and the restored images. The size of the images are respectively 574×768, with $n = 30$; 574×768, with $n = 40$; and 2208×2688, with $n = 128$. [→ Example 4.12]

In this code, we use function res_minio of pyxvis Library. This function computes MINIO restoration algorithm as defined in (4.34). □

The restoration quality is equally as good as the classical methods (see for example [5]), while the computation load is decreased considerably (see comparisons in [12]).

4.7 Summary

In this chapter, we covered the main techniques of image processing used in X-ray testing.

They are:

- Image preprocessing: Noise removal, contrast enhancement, and shading correction.
- Image Filtering: linear and non-linear filtering.
- Edge detection: Gradient estimation, Laplacian-of-Gaussian, and Canny.

- Image segmentation: Thresholding, region growing, and maximally stable extremal regions.
- Image restoration: Minimization of the norm between input and output.

The chapter provided a good overview, presenting several methodologies with examples using real and simulated X-ray images.

References

1. Boerner, H., Strecker, H.: Automated X-ray inspection of aluminum casting. IEEE Trans. Pattern Anal. Mach. Intell. **10**(1), 79–91 (1988)
2. Canny, J.: A computational approach to edge detection. IEEE Trans. Pattern Anal. Mach. Intell. **PAMI-8**(6), 679–698 (1986)
3. Castleman, K.: Digital Image Processing. Prentice-Hall, Englewood Cliffs (1996)
4. Faugeras, O.: Three-Dimensional Computer Vision: A Geometric Viewpoint. The MIT Press, Cambridge (1993)
5. Gonzalez, R., Woods, R.: Digital Image Processing, 3rd edn. Prentice Hall, Pearson (2008)
6. Haralick, R., Shapiro, L.: Computer and Robot Vision. Addison-Wesley Publishing Co., New York (1992)
7. Heinrich, W.: Automated inspection of castings using X-ray testing. Ph.D. thesis, Institute for Measurement and Automation, Faculty of Electrical Engineering, Technical University of Berlin (1988). (in German)
8. Matas, J., Chum, O., Urban, M., Pajdla, T.: Robust wide-baseline stereo from maximally stable extremal regions. Image Vis. Comput. **22**(10), 761–767 (2004)
9. MathWorks: Image Processing Toolbox for Use with MATLAB: User's Guide. The MathWorks Inc. (2014)
10. Mery, D.: Crossing line profile: a new approach to detecting defects in aluminium castings. In: Proceedings of the Scandinavian Conference on Image Analysis (SCIA 2003). Lecture Notes in Computer Science, vol. 2749, pp. 725–732 (2003)
11. Mery, D., Berti, M.A.: Automatic detection of welding defects using texture features. Insight-Non-Destruct. Testing Condit. Monitor. **45**(10), 676–681 (2003)
12. Mery, D., Filbert, D.: A fast non-iterative algorithm for the removal of blur caused by uniform linear motion in X-ray images. In: Proceedings of the 15th World Conference on Non-Destructive Testing (WCNDT–2000). Rome (2000)
13. Mery, D., Filbert, D.: Automated flaw detection in aluminum castings based on the tracking of potential defects in a radioscopic image sequence. IEEE Trans. Robot. Autom. **18**(6), 890–901 (2002)
14. Mery, D., Pedreschi, F.: Segmentation of colour food images using a robust algorithm. J. Food Eng. **66**(3), 353–360 (2004)