Image Analysis Techniques for Industrial Inspection Systems

Przemysław Pietrzkiewicz

Copyright © 2012-2019 Adaptive Vision sp. z o.o.

Contents

Contents i

Introduction 1

1 Image Thresholding 5
1.1 Introduction 6
1.2 Global Thresholding 6
1.3 Threshold Selection 7
1.4 Dynamic Thresholding 15

2 Blob Analysis 19
2.1 Introduction 20
2.2 Region 21
2.3 Elementary Operators 24
2.4 Mathematical Morphology 27
2.5 Topology 33
2.6 Features 36
2.7 Examples 40

3 1D Edge Detection 43
3.1 Introduction 44

3.2 Profile Extraction 45
3.3 Step Edges 48
3.4 Ridges 53
3.5 Stripes 56
3.6 Examples 57

4 2D Edge Detection 61
4.1 Introduction 62
4.2 Image Gradient 62
4.3 Canny Edge Detector 67

5 Contour Analysis 73
5.1 Introduction 74
5.2 Path 74
5.3 Segmentation 75
5.4 Statistical Features 79
5.5 Geometrical Features 83

6 Shape Fitting 87
6.1 Introduction 88
6.2 Lines 89
6.3 Circles 94
6.4 Fitting Approximate Primitives to Images 97
6.5 Examples 98

7 Template Matching 101
7.1 Introduction 102
7.2 Brightness-Based Matching 103
7.3 Edge-Based Matching 111
7.4 Examples 114

Bibliography 115

Introduction

No profit grows where is no
pleasure ta'en; in brief, sir, study
what you most affect.

William Shakespeare

The aim of this work is to discuss a selection of the most popular **image analysis** techniques in the context of **industrial inspection** applications. We will explain the mechanics of each method and demonstrate their applicability (or lack of such applicability) in the industrial setting using real industrial images.

Scope

When selecting the specific set of methods to be discussed in the work, we have decided to focus on methods that meet the following criteria:

- **Direct relation with image analysis** - we will cover the methods that either directly extract information from images, or are designed specifically for further processing of such information.
- **General-purpose character** - we will discuss the methods that may be employed to address a range of needs, as opposed to methods for decoding information represented in any particular format, such as barcode recognition.

Our discussion will commence with two chapters covering extraction and analysis of pixel-precise image objects (**Image Thresholding**, **Blob Analysis**). Later we will cover sub-pixel precise measurements (**1D Edge Detection**) and extraction and analysis of sub-pixel precise contours (**2D Edge Detection**, **Contour Analysis**). We will conclude the survey with two techniques for locating geometric primitives (**Shape Fitting**) and custom predefined image templates (**Template Matching**).

Reference Implementation

All of the methods were evaluated using **Adaptive Vision Studio 4** and all of the results included in the work come from this software. The specific operators implementing the methods discussed in each section are indicated in Reference Implementation boxes, such as the following:

> **Adaptive Vision Studio 4** filter `LenaImage` produces the well known image of Lena Soderberg.

Free editions of the software include full library of the operators and are available at www.adaptive-vision.com.

Conventions

When naming variables, we use lowercase identifiers such as a, $delta$ to denote real and integer numbers, and uppercase identifiers such as R, $Image$ to denote instances of complex types such as euclidean points, segments, regions or images.

Chapter 1

Image Thresholding

> Truly to enjoy warmth, some small part of you must be cold, for there is no quality in this world that is not what it is merely by contrast. Nothing exists in itself.
>
> — Herman Melville

1.1 Introduction

Classification of image pixels into groups sharing some common characteristics is often the very first step of automatic image interpretation. Typically we wish to segment an image into blobs representing the individual objects it contains, so that they can be subject to measurements or any other mean of inspection.

Usually trivial for the human mind, unsupervised **Image Segmentation** is far from straightforward in general case. The available methods vary in complexity and principles, taking into account various image parameters such as color, brightness, gradient, texture or motion.

In the industrial setting it is often the case that the image content can be clearly divided into background (e.g. the surface of conveyor line or inspection station) and foreground (e.g. the objects being inspected). Such simple, binary pixel classification is called **Image Thresholding**.

1.2 Global Thresholding

Basic thresholding operator simply selects the pixels of intensity within a predefined range. If we interpret the results as a binary image with black pixels denoting the background and white pixels denoting the foreground, the operation applied to an image I computes the result B as follows:

$$B[i,j] = \begin{cases} 1 & \text{if } minValue \leq I[i,j] \leq maxValue \\ 0 & \text{otherwise} \end{cases}$$

Figure 1.1 demonstrates example results of thresholding the same image with different range of foreground intensities.

Global thresholding is *global* in that it evaluates each pixel of the image using the same foreground intensity range. As such, it requires not only that the background is consistently darker (or brighter) than foreground, but also that the lightning is reasonably uniform throughout the entire image.

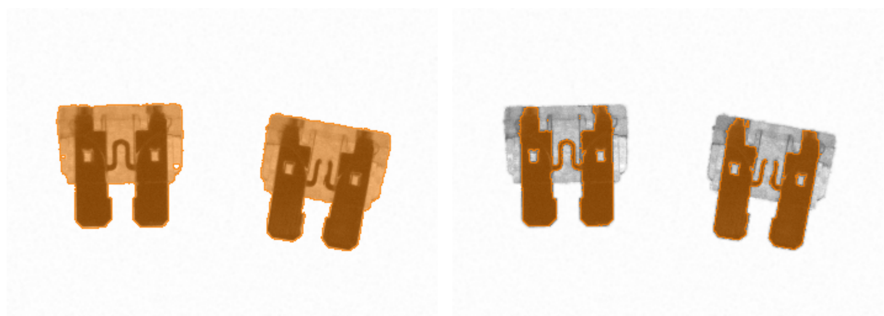


Figure 1.1: Results of global thresholding with different threshold values - pixels identified as foreground marked in orange.

The importance of uniform (in space) and constant (in time, when a series of images is analyzed) lightning for successful application of automatic visual inspection is paramount. Whenever bad lightning conditions disrupt work of a technique, we should try to amend the lightning first, and only if this is not possible we should move to adjusting the algorithm.

That being said, numerous methods were developed to allow successful thresholding despite the lightning imperfections.

1.3 Threshold Selection

If the lightning is reasonably uniform throughout the image, but changes over time (which is usually the case whenever the system is not fully isolated from the sunlight) the threshold values should be adjusted accordingly. As the system should be essentially unsupervised in operation, we need to employ a technique that will allow us to determine the feasible threshold automatically, given only the image to be thresholded.

Applying such technique would also eliminate the bias introduced by manual adjustment of the threshold parameters - usually there is a range of feasible threshold values and the extracted objects appear smaller or bigger depending on the selected value.

Automatic threshold selection has been subject to extensive research and a rich set of different methods has been developed. A survey[1] by Sezgin and Sankur mentions 31 different methods of automatic selection of global thresholding values. We will demonstrate a selection of techniques particularly popular in the industrial applications.

The distribution of pixel intensities is an important source of information about the applicability of global thresholding and the possible threshold values. Because of that we will present histogram of pixel intensities along each example in this section.

We will demonstrate the strengths and weaknesses of individual methods using a set of industrial images demonstrated in **Figure 1.2**. As most of these images are used more than once, we have decided to display them collectively for brevity, in later section presenting solely the thresholding results.

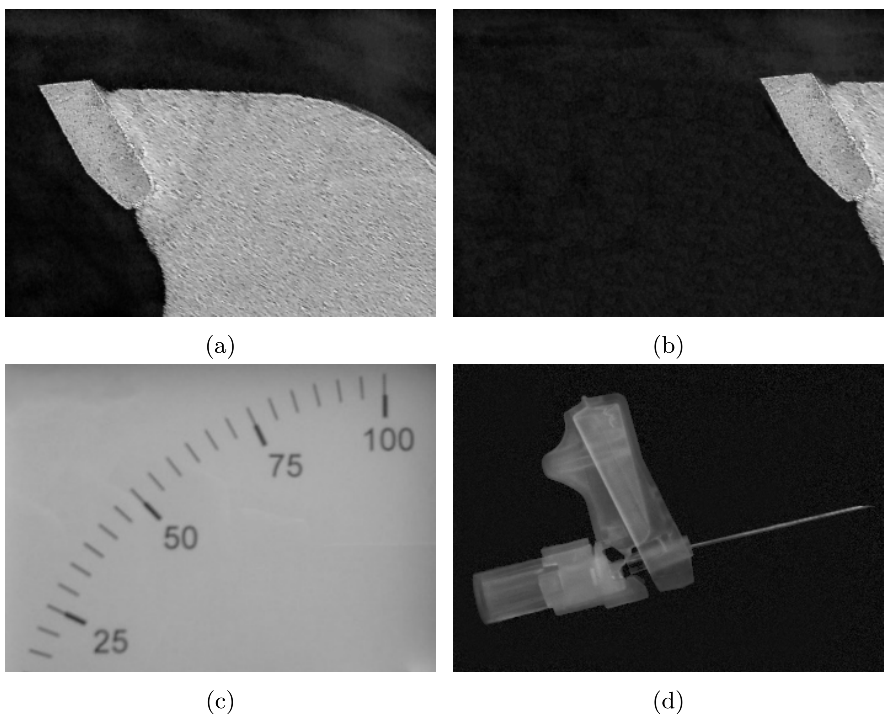
Mean Brightness

As long as both background and foreground are consistent in brightness and occupy similar proportion of the image space, we may expect that the average image intensity will lie somewhere between the intensities of objects and background and as such would be a feasible threshold value.

In **Figure 1.3** we can see an image for which this method performs correctly. Well separated background and foreground intensities appear as two significant modes in the image histogram. The modes are similar in size, which is a consequence of roughly even distribution of background and foreground in the image space.

Unsurprisingly, the average pixel brightness (denoted with vertical line in the histogram) fits between the two modes and allow for accurate thresholding.

Unfortunately the accuracy of this method quickly drops as the disproportion between background and foreground increases. **Figure 1.4** demonstrates an example for which the method fails, even though the histogram modes are still



(a) (b) (c) (d)

Figure 1.2: Four images used to benchmark threshold selection methods.

well-separated and the range of feasible threshold values is trivial to read from the histogram.

This makes the method in its basic form not advisable for most of the industrial applications, although its shortcomings may be addressed using edge detection, which we will inspect in detail in later chapter. If we compute the average brightness using only the pixels in fixed neighborhood of edges separating objects from background, we may assume that roughly the same amount of background and foreground pixels will be taken into account.

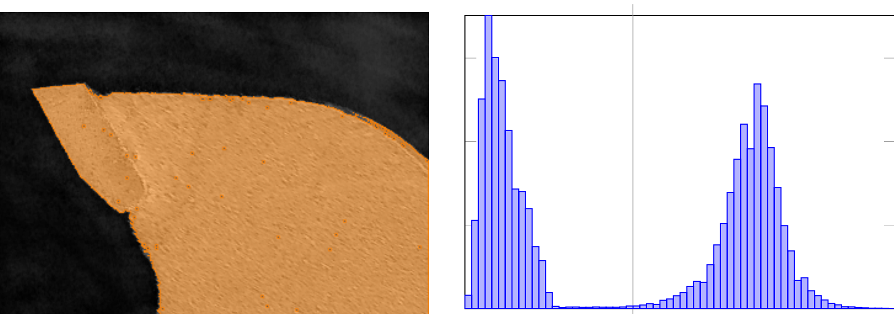


Figure 1.3: Example image successfully thresholded using mean brightness as the threshold value.

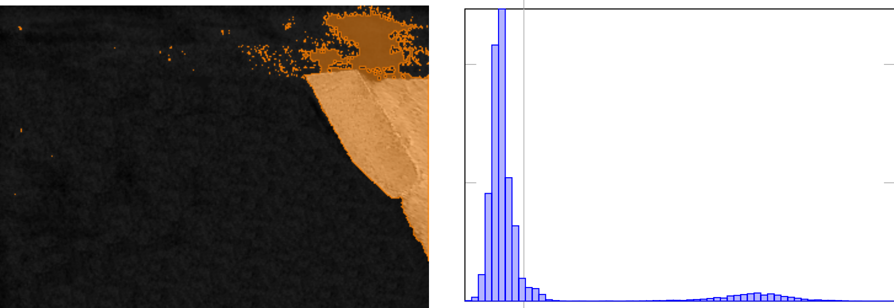


Figure 1.4: Example image for which mean brightness is not a feasible threshold value.

Histogram Shape Analysis

In the previous section we have seen two examples of images having bimodal histograms with a clear valley between two modes corresponding to the range of feasible threshold values. Some of the popular threshold selection methods look for this valley algorithmically - either directly or indirectly, analyzing the shape properties of image histogram.

In one of the first papers[2] written on the threshold selection problem Prewitt and Mendelsohn proposed to smooth the image histogram iteratively until only two local maxima are preserved, and then select the threshold value as

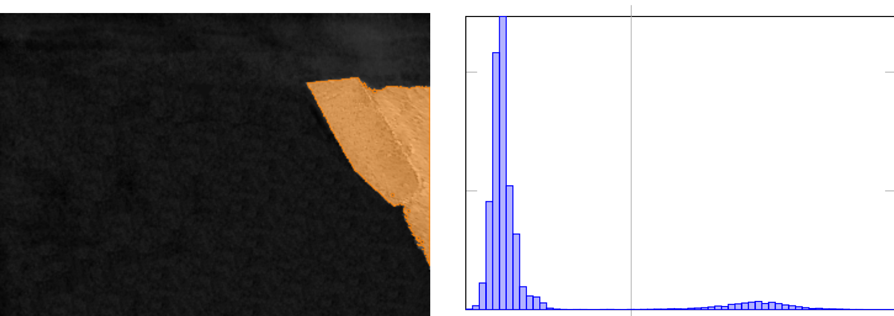


Figure 1.5: Threshold selection using Intermodes method.

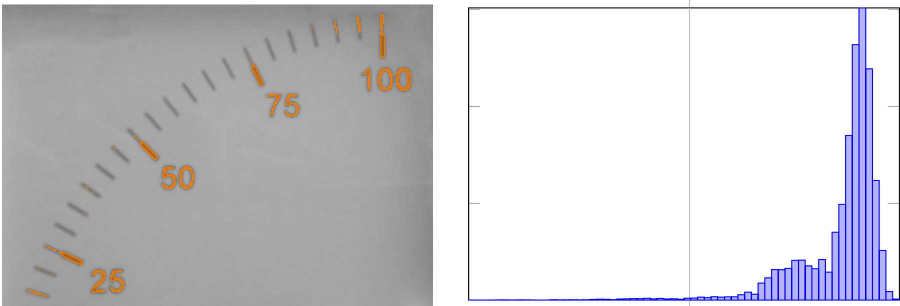


Figure 1.6: Threshold selection using Intermodes method.

a mean of this two remaining maxima. **Figure 1.5** demonstrates a successful application of this method to an example for which the mean brightness method failed.

Intermodes method performs well as long as the image histogram is essentially bimodal, but yields unstable results whenever this is not the case. **Figure 1.6** demonstrates an example which at the first sight seems to have an obvious threshold value, but in fact has unimodal histogram with hardly any peak corresponding to the foreground. Unsurprisingly, the method fails on such image.

Entropy

Entropy of a distribution (e.g. distribution of the pixel brightness, i.e. image histogram) is a quantitative measure of its *disorder* - it is high when the fractions of pixels taking the individual intensities are similar throughout the range of possible pixel values[1] and low when certain values are overrepresented in the image.

The most commonly measure of entropy is the one proposed by Shannon, in which the entropy of a distribution D is defined as:

$$Entropy(D) = -\sum_{(v,f)\in D} f \log_2 f$$

where each element of the distribution (v, f) represents that a value v is taken by the fraction f of the data, i.e. $\sum_{(v,f)\in D} f = 1.0$.

Numerous attempts were made to employ analysis of entropic properties of the image intensity distribution to threshold selection, from early works of Pun to more recent investigations on applications of fuzzy entropy measures. We will demonstrate the method proposed[3] by Kapur, Sahoo and Wong which is indicated[1] as well-performing by Sezgin and Sankur.

In this technique the image intensity distribution is split into foreground and background distributions at the intensity level k for each possible value of k. Then the entropy of both distributions is computed and the k which yields the largest sum of two entropies is selected as the threshold value.

The rationale for such schema lies in the fact that after the thresholding both foreground and background pixels will be set to a constant value[2], thus the entropy of both distributions will be reduced to 0. Therefore the threshold value that maximizes the entropies of two classes can be thought to imply the biggest reduction of disorder in image (at least in terms of intensity distribution).

[1] Such situation represents high disorder because the pixel values are uniformly scattered over the entire domain of pixel intensities.

[2] E.g. 0 for background and 255 for foreground.

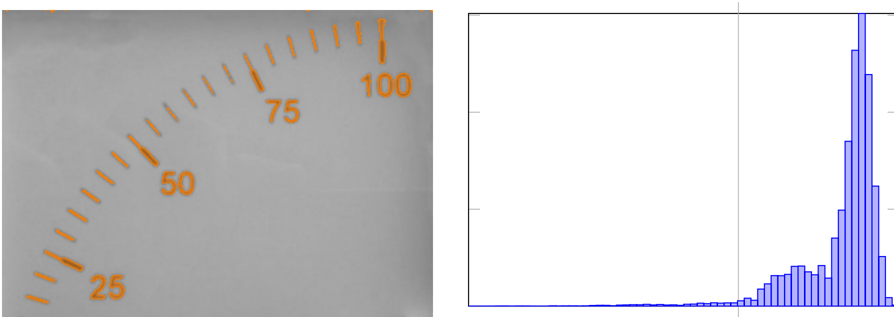


Figure 1.7: Threshold selection using Entropy method.

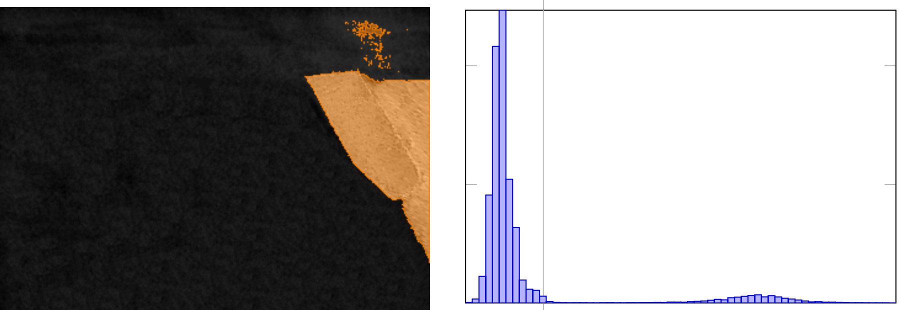


Figure 1.8: Threshold selection using Entropy method.

Such approach to threshold selection can yield good results on images for which other methods fail, as demonstrated in **Figure 1.7**, but it may also fail spectacularly on apparently simple images with clear, bimodal histogram, as demonstrated in **Figure 1.8**. For this reason we would discourage use of this technique in the industrial setting.

Clustering

Threshold selection problem may be also formulated in terms of clustering - indeed, our aim is to divide the full intensity spectrum into two clusters, foreground and background intensities, separated by the threshold value. In this section we will demonstrate two techniques that follow this interpretation of the problem.

K-Means

Well known general-purpose K-Means algorithm iteratively computes the means of current set of K clusters, and reassigns the elements being clustered, each one to the cluster represented by the nearest of means computed in this iteration.

This idea was applied[4] to threshold selection problem by Ridler and Calvard. The algorithm maintains two clusters containing complementary parts of the intensity range. At each step mean brightness of the pixels in each cluster is computed and pixels are reassigned to the cluster of nearest mean.

It is worth noting that even though the authors proposed an iterative scheme of computation, it is perfectly feasible to perform a brute force search over all possible threshold values and select the one that fits halfway between means of the induced clusters; especially in the common case of uint8 industrial images of only 255 possible intensity levels. If we precompute the image histogram and maintain the running averages of the clusters, we may process each threshold value in constant time.

Otsu Method

One of the first clustering approaches was proposed[5] by Otsu, who described a method that selects the threshold value that maximizes the between-class variance between foreground and background intensities:

$$|F| \cdot |B| \cdot (\overline{F} - \overline{B})^2$$

where $|F|$ and $|B|$ denote, accordingly, the number of foreground and background pixels while $\overline{F}$ and $\overline{B}$ denote their mean values.

Both methods of clustering-based threshold selection yield similar results for the set of benchmark images used in this chapter. As the methods do not explicitly look for a valley between histogram modes, they can give stable and correct results for some images for which histogram shape analysis methods fail. Unfortunately, similarly to the entropic method we have already seen, the methods fail in some cases for which simple methods work correctly, as demonstrated in **Figure 1.9**.

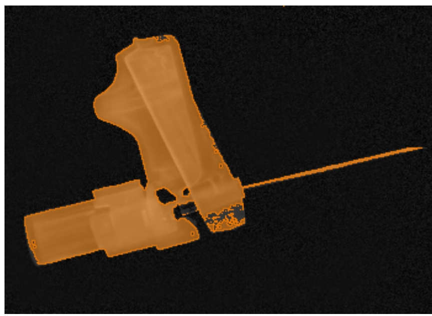
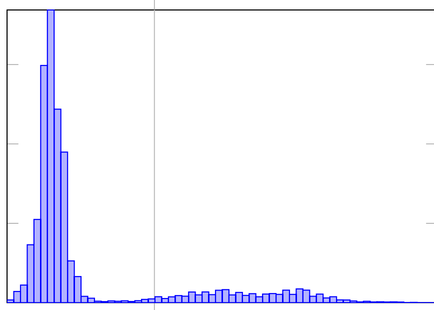



Figure 1.9: Threshold selection using Otsu method.

Summary

As we have demonstrated, automatic threshold selection is not an easy task in general case. As long as the image has clear, bimodal histogram, we may relay on the accuracy of simple histogram shape-analysis based methods, but when this is not the case things get more complicated.

If the system is supposed to operate at high levels of reliability, it would be prudent to use global thresholding with automatic threshold selection only if we can guarantee the conditions in which a method of our choice performs correctly. Often this will not be possible, in which case we should consider using dynamic thresholding (discussed in the next section) or discarding the thresholding methods at all in favor of **Contour Analysis** or **Template Matching**.

> **Adaptive Vision Studio 4** filter `SelectThresholdValue` implements the following threshold selection methods: HistogramIntermodes, ClusteringKittler, ClusteringKMeans, ClusteringOtsu, Entropy.

1.4 Dynamic Thresholding

When the lightning in the scene is uneven to the point where image foreground intensity in dark parts of the image is at the same level as the background intensity in bright sections - or, in other words, intensity ranges of background and foreground are overlapping - it is clear that global thresholding cannot be applied.

Figure 1.10: .

An example of such problem is illustrated in **Figure 1.10**. As we can see, bad lightning setup makes left bars of the barcode appear brighter than the background in the right part of the barcode. Key point in overcoming this issue lies in an observation that the barcode, even under bad lightning, is still *locally* darker than the background in its entirety.

This is illustrated in **Figure 1.11**, where we plot the 1D profile of the barcode extracted along the scan line marked in the image and the same profile smoothed with running average operator of with 10.

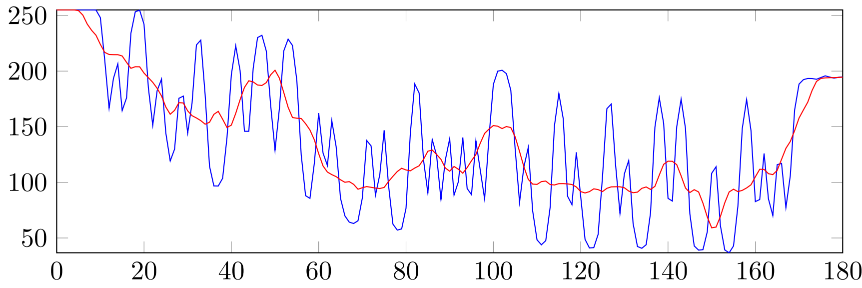


Figure 1.11: Brightness profile of the barcode image in blue, smoothed profile in red.

Therefore, if we define the threshold value in relation to mean local brightness at each location, we can get accurate results despite bad lightning conditions.

Dynamic Thresholding classifies the pixels of image I in relation to image A representing local brightness means:

$$B[i,j] = \begin{cases} 1 & \text{if } minValue \leq I[i,j] - A[i,j] \leq maxValue \\ 0 & \text{otherwise} \end{cases}$$

The image of local averages may be obtained using smoothing operator. Depending on the specific application we may prefer to use different smoothing methods. In practice mean blur with box kernel is frequently used due to its efficiency and despite its anisotropy. Gaussian operator is isotropic, yet slower alternative.

Figure 1.12 demonstrates failed global thresholding attempt (with lowest threshold value that includes whole bar area in the result) and the results of dynamic thresholding of the image using mean blur with box kernel.

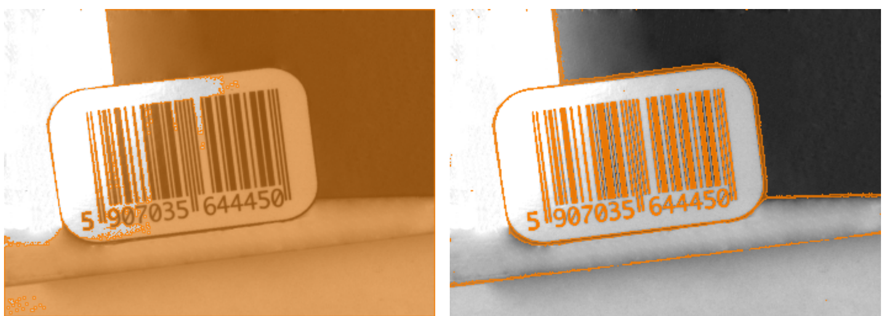


Figure 1.12: Results of global thresholding and dynamic thresholding of the barcode picture.

> **Adaptive Vision Studio 4** filters `ThresholdImage_Dynamic` and `ThresholdToRegion_Dynamic` implement the dynamic thresholding method using mean average with box kernel.

Chapter 2

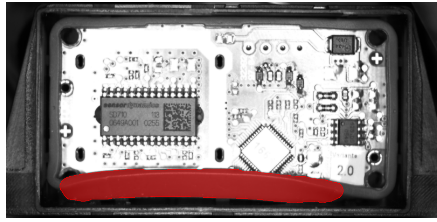
Blob Analysis

> Mr Herbert was cleaning a cupboard when he found the blob of glue. His girlfriend noticed that it looked similar to Homer Simpson, and he decided to try to sell it on eBay.
>
> http://web.orange.co.uk

2.1 Introduction

In the previous chapter we have been looking into methods that allow us to extract pixel-precise regions corresponding to the objects present in the image. The obtained regions can be and usually are subject to inspection - measurements, classification, counting, etc. Such analysis of pixel-precise shapes extracted from image is called **Blob Analysis**, Region Analysis or Binary Shape Analysis.

Blob Analysis is a fundamental technique of image inspection; its main advantages include high flexibility and excellent performance. Its applicability is however limited to tasks in which we are able to reliably extract the object regions (see **Template Matching** for an alternative). Another drawback of the technique is pixel-precision of the computation (see **Contour Analysis** for a subpixel-precise alternative).



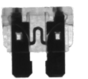


Figure 2.1: Example Blob Analysis applications - detection of excessive rubber band segment and disconnected fuses.

A typical Blob Analysis-based solution consists of the following steps:

1. **Extraction** - firstly, the region corresponding to image objects is extracted from the image, usually by means of **Image Thresholding**.
2. **Processing** - secondly, the region is subject to various transformations that aim at enhancing the region correspondence to the actual object or highlighting the features that we want to inspect. In this phase the region is often split into connected components so that each one can be analyzed individually.
3. **Feature Extraction** - in the final part the numerical and geometrical features describing the refined regions, such as its diameter, perimeter,

compactness, etc. are computed. Such features may be the desired result themselves, or be used as discriminants for region classification.

As **Image Thresholding** has already been discussed in the previous chapter, this chapter will focus entirely on two latter steps. We will commence with a demonstration of the data structure that we will use for representation of pixel-precise shapes and proceed to discussion of morphological and topological transformations that may be performed on such shapes. After that we will review the numerical and geometrical features of binary shapes that are particularily useful for the needs of visual inspection and conclude the chapter with a handful of example Blob Analysis applications.

2.2 Region

Region is the fundamental data type for representation of pixel-precise binary shapes. Formally, it may be defined as follows:

Region is **any** subset of image pixel locations.

As follows from this definition, a region may represent any pixel-precise shape present in an image, connected or not, including empty region and full region. Image Thresholding operations discussed in the previous chapter return a single region - possibly representing a number of image objects.

Data Representation

The actual representation of a region in computer memory does not affect the theory of **Blob Analysis** but has important practical implications. Typically, the decision on the data representation boils down to the trade-off between memory efficiency of the data storage and computational efficiency of the operations that we intend to perform on data instances.

Binary Image

One trivial representation of a region would be a binary image, each of its pixels having a value of 0 (not-in-region) or 1 (in-region). Such representation is quite verbose, as each region (even empty region) consumes an amount of memory corresponding to the size of the original image. On the other hand,

this representation allows $O(1)$ lookup time for determining whether a pixel belongs to a given region.

Run-Length List

We could reduce the memory consumption using a classic data compression technique: Run-Length Encoding. In this technique consecutive, uniform sections (runs) of data are stored as tuples $(value, length)$. In case of binary values, we may use an alternative form in which the runs of *ones* are represented as tuples $(position, length)$ and the runs of *zeros* are represented implicitly as the complement of *ones*. We may use the latter form to represent horizontal runs of region pixel locations as tuples $(x, y, length)$, where x and y denote the coordinates of the first pixel of the run.

Such representation does not allow for $O(1)$ random-pixel access anymore, but as long as the list of pixel runs is sorted, we can achieve $O(log(R))$ pixel lookup time, R denoting the number of pixel runs. In return this representation allows to perform various operations (such us region intersection or moment computation) in time dependent on the number of runs rather than number of pixels; which yields significant speed-up in typical applications.

As to memory efficiency, the results of a simple benchmark are presented in **Table 2.1**. We took into account three representations, each applied to store four regions extracted from 250x200 images.

- **Binary Image (uint8)** - a variant of Binary Image representation in which each pixel is stored as 0 or 1 value of 8-byte integer value. Although suboptimal, 8-byte per pixel is prevalent pixel depth for such applications because of the low-level details of memory access.

- **Binary Image (bit)** - a variant of Binary Image representation in which each pixel is stored as 0 or 1 value of a single bit.

- **Run-Length Encoding** - we assumed that each element of the $(x, y, value)$ tuple is stored using 16-bit integer type, which accumulates to 6-bytes per pixel run memory usage.

	Image (uint8)	Image (bit)	RLE
	50000	6250	0
	50000	6250	978
	50000	6250	6276
	50000	6250	75102

Table 2.1: Number of bytes consumed by different region representations.

Region Dimensions

In our reference implementation, **Adaptive Vision Studio 4** , regions are represented using the run-length encoding described above, with one slight extension: each region stores two additional integers representing its reference dimensions: width and height.

These are usually the dimensions of image the region was extracted from and serve two purposes. For one thing, they allow meaningful display of a region in the context of image it refers to; for other thing - they conveniently allow to define a complement of a region. Formally, the finite dimensions of the region space allows to distinguish between three types of pixels: set, unset and undefined (outside the region dimensions), i.e. corresponding to undefined image pixels that we have no information about.

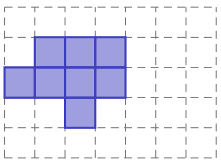


Figure 2.2: Region of dimensions: 7 (width), 5 (height)

2.3 Elementary Operators

In this section we will introduce six elementary operations that can be performed on regions. Four of them refer to the set nature of regions, two further are defined in relation to its spatial properties.

In the next section we will use these building blocks to define powerful transformations from the field of Mathematical Morphology.

Set Operators

Applicability of basic set operators to region processing follows directly from the definition of region.

Union

Union of two regions is a region containing the pixels belonging to either, or both of the input regions, as demonstrated in **Table 2.2**.

Intersection

Similarly, intersection of two regions is a region containing the pixels belonging to both of the input regions, as demonstrated in **Table 2.3**.

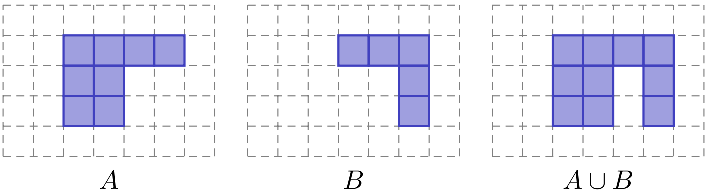


Table 2.2: Union of two regions

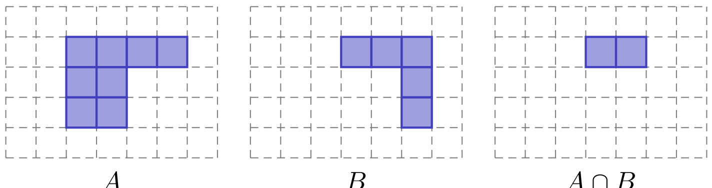


Table 2.3: Intersection of two regions

Difference

Last binary operation in this group is difference, yielding the pixels belonging to first region, but not to the second region. Thus, this operation is not commutative, contrary to intersection and union.

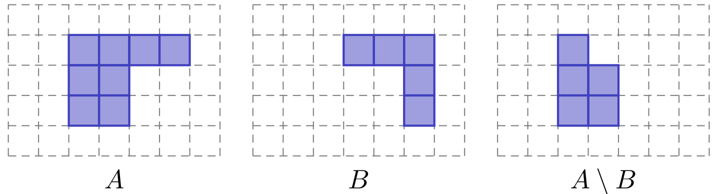


Table 2.4: Difference of two regions

Complement

The only unary set operator, complement, is also applicable to region; however industrial implementations differ in its interpretation. We will follow the

way of our reference implementation, where complement is easy to define as each region stores the dimensions of its finite reference space.

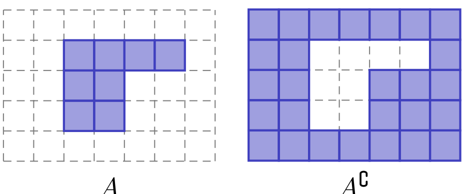


Table 2.5: Complement of a region

Spatial Operators

Two further operators refer to spatial properties of region. Naturally, there are far more spatial operators than can be defined for region; for now we introduce only two that are necessary to define morphological operators discussed in the next section.

Translation

Translation of a region shifts its pixel coordinates by integer vector.

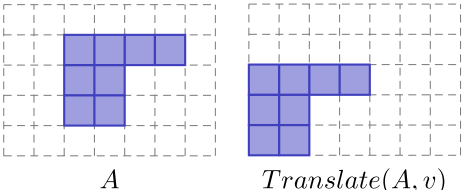


Table 2.6: Translation of a region by vector -2,1.

Reflection

Reflection mirrors a region over a location (origin). This operation will be particularly useful for processing morphological kernels, which we will discuss in the next section.

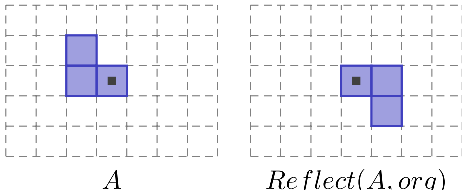


Table 2.7: Reflection of a region, its origin marked with a black square.

2.4 Mathematical Morphology

Mathematical Morphology, born in 1960s and rapidly developing ever since, is both a theory and a technique for processing spatial structures. Soille described[6] the field as being *mathematical* in that it is built upon set theory and geometry, and being *morphology*[1] in that it aims at analyzing the shape of objects.

In most general case, Mathematical Morphology applies to any complete lattice. We will concentrate on its application to region processing. In this context Mathematical Morphology can be looked at as a set of techniques that alter a region by probing it with another shape called kernel or structuring element.

Kernel

Kernel in Mathematical Morphology is a shape that is repeatedly aligned at each position within the dimensions of the region being processed. At each such alignment the operator verifies how the aligned kernel fits in the region (e.g. if the kernel is contained in the region in its entirety) and depending on the results includes the location in the results or not.

As kernel is pixel-precise binary shape itself, it can be represented as a region together with integer coordinates of its origin. Specifying the origin is important, as it is the position that will be aligned against the region being processed.

[1] From Greek *morphe* meaning form.

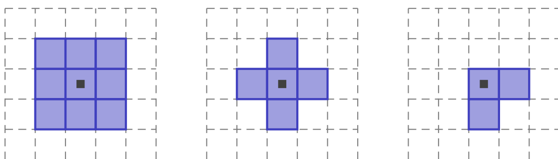


Table 2.8: Example kernels for morphological operations.

Dilation

First morphological operation that we are going to discuss is dilation. In this operator the kernel aligned at each position within the region dimensions needs to overlap with at least one pixel of the input region to include this position in the result:

$$Dilate(R, K) = \{[p_x, p_y] | R \cap Translate(K, [p_x, p_y]) \neq \emptyset\}$$

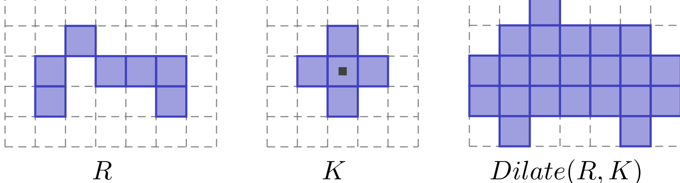


Table 2.9: Dilation of a region

If we decompose the kernel into its individual pixels we may observe that each such pixel $[k_x, k_y] \in K$ contributes a copy of the region translated by $[-k_x, -k_y]$ into the result. Therefore we may also define the dilation operator as follows:

$$Dilate(R, K) = \bigcup_{[k_x, k_y] \in K} Translate(R, [-k_x, -k_y])$$

Dilation effectively expands the region, the magnitude and direction of the expansion depending on the kernel being used. The operator is commonly used to join disconnected components of a region. Dilating a region by circular kernel

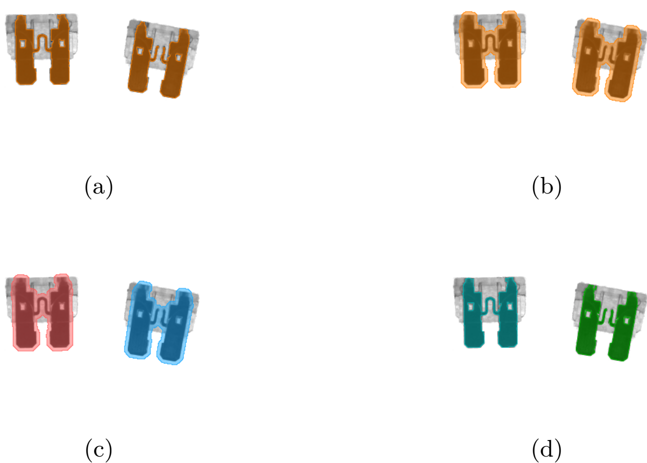


Figure 2.3: Dilation, extraction of connected components and intersection applied to split a region representing metal parts of the fuses (a) into components representing individual fuses (d).

of radius r will expand the region uniformly in each direction up to distance of r pixels, effectively joining region components separated by less than $2r$ pixels. One possible application is demonstrated in **Figure 2.3**.

In this example we process a region representing metal parts of two fuses. As one of the fuses is burned out, the region contains three connected components. To split it into two connected components, each representing an individual fuse, we may perform the dilation before extracting the region components and intersect the resulting regions with the original one to preserve their original shape.

Erosion

Erosion is a shrinking counterpart of dilation. This operator requires that the aligned kernel is fully contained in the region being processed:

$$Erode(R, K) = \{[p_x, p_y] | Translate(K, [p_x, p_y]) \subseteq R\}$$

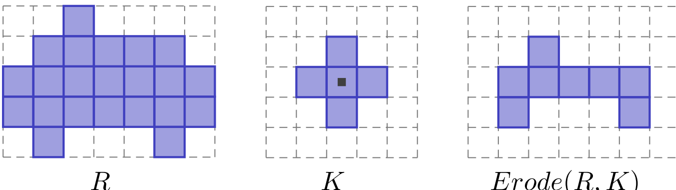


Table 2.10: Erosion of a region

Similarly to dilation, we may also formulate erosion in terms of kernel decomposition. In this case each pixel of the kernel $[k_x, k_y] \in K$ also contributes the shifted copy of a region, but a position must be contained in all such contributions to be included in the results:

$$Erode(R, K) = \bigcap_{[k_x, k_y] \in K} Translate(R, [-k_x, -k_y])$$

The operations of dilation and erosion are closely related, but it is important to note that they are not inverse[2] of each other, i.e., erosion of a region does not necessarily *cancel out* previously applied dilation; counterexample being presented in **Table 2.9** and **Table 2.10**. Quite contrary, consecutive application of dilation and erosion is extremely useful operation and will be discussed soon.

Although dilation is not an inverse of erosion, another relation between the operations holds - they are duals of each other, meaning that dilation of a region is equivalent to erosion of its background (complement), and conversely.

$$Erode(R, K) = Dilate(R^{\mathsf{C}}, K)^{\mathsf{C}}$$

Closing

Before we define the next operator, let us get back for a moment to the dilation operator. As we remember, dilation expands the region in the way defined by the structuring element. It is worth noting that during this expansion small

[2] Actually neither of these operation has an inverse, as such operation would have to magically guess where the lone pixels lost in erosion or holes filled in dilation were located.

holes and region cavities may get completely filled in. This effect is worth attention as filling gaps of a region[3] is a common need in industrial inspection.

Unfortunately, dilation does not address this need precisely - the missing parts gets filled in, but also the region boundaries are expanded. It would be more convenient to have an operator that avoids the second effect while keeping the first.

The closing operator addresses this need by dilating the region and eroding it right after that:

$$Close(R, K) = Erode(Dilate(R, K), Reflect(K))$$

Initial dilation fills in the region gaps and the succeeding erosion brings the expanded region back to its original dimensions (but does not restore the gaps that were completely filled in).

It is worth noting that we use the reflected kernel for the second operation - if we recall that dilation may be formulated as a union of translations corresponding to individual pixels of the kernel ($\bigcup_{[k_x, k_y] \in K} Translate(R, [-k_x, -k_y])$), it is clear that we need to use the opposite translations to keep the region in its position.

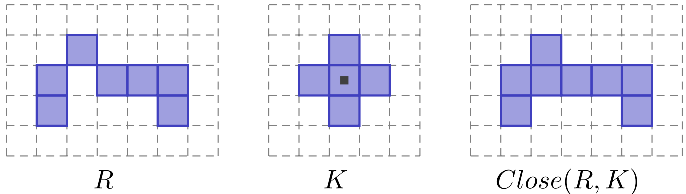


Table 2.11: Closing of a region

Closing is commonly applied whenever the extracted region contains gaps or cavities that should be filled in, an example of such application is demonstrated in **Figure 2.4**.

[3] Which could be introduced for instance by local glare of the lightning affecting the results of thresholding.

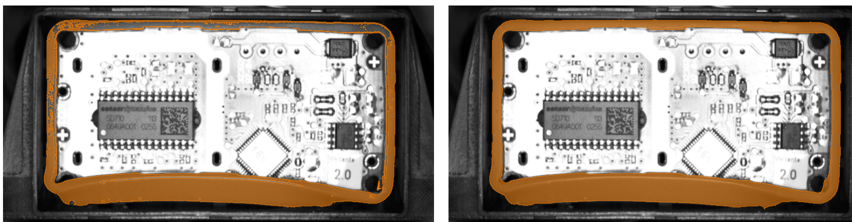


Figure 2.4: Closing operator used to fill gaps in a region.

Opening

Another useful morphological operator is obtained by interchanging the order of operators that closing is composed of. The opening operator firstly erode a region and then dilates the result:

$$Open(R, K) = Dilate(Erode(R, K), Reflect(K))$$

The effect of such composition is dual to the closing operator that we recently discussed. The initial erosion shrinks the region removing its isolated pixels and small branches, while the successive dilation brings it back to original dimensions, but cannot restore the parts that vanished completely during erosion.

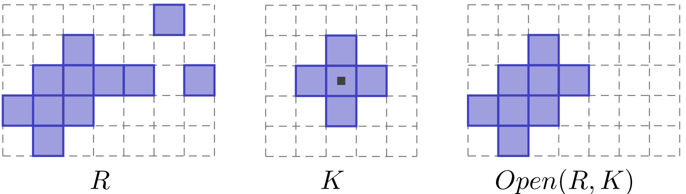


Table 2.12: Opening of a region

The opening operator may be applied to remove salt noise in the region or to eliminate its thin parts. Opening a region using a circular kernel of radius r will remove all segments of the region that have less than $2r$ pixels in width (and keep the other parts intact). An example application is demonstrated in **Figure 2.5**.

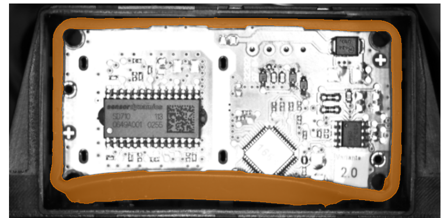
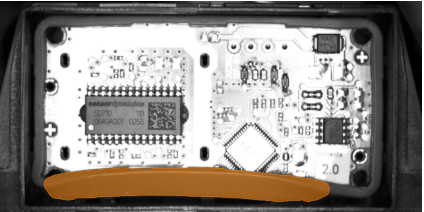
 

Figure 2.5: Opening operator used to determine excessively wide section of the rubber band.

Basic morphological operators described in this section are available as **Adaptive Vision Studio 4** filters:

- `DilateRegion`, `DilateRegion_AnyKernel`
- `ErodeRegion`, `ErodeRegion_AnyKernel`
- `CloseRegion`, `CloseRegion_AnyKernel`
- `OpenRegion`, `OpenRegion_AnyKernel`

The filters with *_AnyKernel* suffix allow to perform the operation using arbitrary kernel, while their counterparts allow to choose from a set of predefined, hard-coded kernels.

2.5 Topology

We have already seen operations defined in the context of set nature of the region as well as operations concerned with its spatial properties. The last set of transformations that we will discuss is built upon topological concepts such as neighborhood, connectivity or boundaries.

Connectivity

Pixel connectivity defines the conditions in which we say that two pixels are *connected* and as such is a key concept in the context of topological transformations. There is a well known paradox inherently associated with the definition of binary shapes connectivity on square grids[4].

[4] On triangular grids as well, but not on hexagonal ones.

To demonstrate the paradox let us consider a closed not self-intersecting curve. Jordan Curve Theorem (and our common sense) requires that such curve should divide the space in which it lies into exactly two parts - interior and exterior. The paradox of pixel connectivity lies in the fact that this requirement is not met under either of two reasonable definitions of pixel connectivity.

One possible definition of connectivity is 4-connectivity in which a pixel is consider connected to the pixels it shares an edge with, as demonstrated in **Table 2.13**. Under this definition the curve on the right splits the space into three, rather than two connected components.

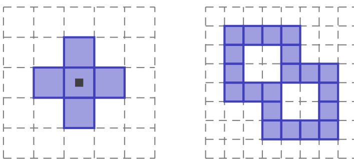


Table 2.13: 4-connectivity kernel and a curve demonstrating a violation of the Jordan curve property.

Another option to consider is 8-connectivity in which a pixel is considered connected to the pixels it shares a corner with, as demonstrated in **Table 2.14**. Unfortunately in this case it is possible to construct a closed curve that does not split the space at all, i.e. curve the interior of which remains connected to the exterior, as demonstrated on the right.

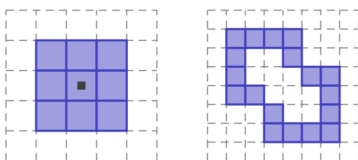


Table 2.14: 8-connectivity kernel and a curve demonstrating a violation of the Jordan curve property.

Rosenfeld proposed[7] to address this problem by using different connectivities for foreground and background, which yields two feasible configurations:

- 4-connectivity for foreground, 8-connectivity for background
- 8-connectivity for foreground, 4-connectivity for background

From now on we will use the terms 4-connectivity and 8-connectivity in the context of foreground connectivity, quietly assuming that the other type of connectivity is used for background.

Connected Components

The necessity of splitting a region into its connected components (also referred to as *blobs*) occurs naturally whenever the image contains a number of objects and single thresholding is used to extract one region collectively representing all of them. We have already seen an example of such proceeding in **Figure 2.3**.

Let us begin with a remark on more general problem of computing the connected components of any graph. In such case there are two ways one can follow: we can either traverse the components of the graph one component at a time (using either breadth-first or depth-first search) or go through the edges of the graph maintaining and updating a Union-Find structure representing our knowledge of connected components in graph (i.e. determining all connected components at the same time).

As we represent regions as lists of pixel runs, the problem boils down to identification of the connected components of a set of pixel runs. Both of the approaches described above can be adapted to do so. Careful implementation of the DFS/BFS based technique performs in $O(R)$ time complexity, while the Union-Find based solution works in almost indistinguishable in practice complexity of $O(R \log^* R)$.

In either case to achieve target complexity it is crucial to carefully go through the region point runs determining the neighbors of each run (either building graph to which we will apply DFS/BFS or updating the Union-Find structure). Splitting the list of point runs into separate lists for each row within the region dimensions allows to do that in $O(R)$ time.

Region Holes

The intuitive concept of a region hole may be formalized as follows:

Region hole is a connected component of region complement that is not adjacent to the boundaries of the region dimensions.

An algorithm for extraction of the region holes may be derived directly from this definition. One detail we should remember about is to use the background connectivity when computing the connected components of the region complement, as discussed before.

> **Adaptive Vision Studio 4** filter `SplitRegionIntoBlobs` extracts an array of region connected components while `RegionHoles` returns an array of region holes computed as described above. Other method of region splitting is available as filters `SplitRegionIntoComponents` and `SplitRegionIntoExactlyNComponents`.

2.6 Features

Once we have acquired a region that accurately represents the object that we intend to analyze, we may proceed to extraction of the region features. Features of two-dimensional shapes (discreet regions as well as continuous polygons) may be organized into two groups: statistical and geometrical.

Statistical features of a shape are built upon statistical concepts such as mean or variance and may be computed directly from the coordinates of region pixels or polygon points, disregarding any spatial relations between them. Statistical shape features include, among others, its area, mass center and orientation (i.e. direction of the principal axis of inertia).

Geometrical features are defined in the context of spatial relations between pixels or points contained in the shape. Some of the features, such as circularity factor, are numeric properties, others, such as smallest bounding circle, take form of geometric primitives.

Almost all of the shape features of both kind that we are going to cover are equally applicable to pixel-precise regions and subpixel-precise polygons, which we will discuss in detail in the **Contour Analysis** chapter. To avoid duplication, in this section we will focus only on few **region-specific features** and region-specific details of shape feature extraction.

Statistical Features

Statistical features of two-dimensional shapes may be conveniently generalized as so called moments. Each moment is a numeric shape feature that sums a simple function of pixel (or point) coordinates over every pixel (or point) contained in the shape.

Formally, we distinguish two types of moments, raw and central, the latter of which considers the coordinate arguments of the function in relation to the average (denoted $\overline{x}$, $\overline{y}$) of the appropriate pixel coordinate, thus achieving translation-invariance. For a given region R, its raw and central moments, are defined, respectively, as:

$$m_{p,q} = \sum_{(x,y)\in R} x^p y^q$$

$$c_{p,q} = \sum_{(x,y)\in R} (x-\overline{x})^p (y-\overline{y})^q$$

where each (p,q) for natural $p \geq 0$, $q \geq 0$ defines different moment. For instance, $m_{0,0}$ equals $\sum_{(x,y)\in R} 1$ and therefore computes the area of the region. $m_{1,0}$ computes the sum of x-coordinates of region pixels and so on.

Both raw and central moments may be normalized, i.e. divided by the area of R, to achieve scale-invariance:

$$m'_{p,q} = \frac{1}{a} m_{p,q}$$

$$c'_{p,q} = \frac{1}{a} c_{p,q}$$

where a denotes the area (number of pixels) of R. For instance, $m'_{1,0}$ computes the average of pixel x-coordinates, i.e. $\overline{x}$ which together with $m'_{0,1}$ form the mass center of R.

As previously indicated, we will discuss applications of these statistics in due course. The region-specific aspect of moment extraction that should be stressed here is that the region moments can be computed directly from their definition, as the number of region pixels is finite; as opposed to infinite number of points contained in continuous polygon.

Moreover, low-order moments may be calculated very efficiently due to the RLE region representation that allow us to process a whole run of pixels in constant time, achieving complexity $O(r)$, where r denotes the number of region pixel runs.

> The following **Adaptive Vision Studio 4** filters extract the statistical features of a region: `RegionArea`, `RegionElongation`, `RegionMassCenter`, `RegionMoment`, `RegionOrientation`, `RegionEllipticAxes`.

Geometrical Features

Contours

The contour of a region is a sequence of points defining its boundary (or an array of such sequences in case of disconnected region), as demonstrated in **Figure 2.6**.

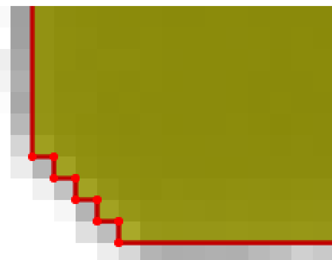


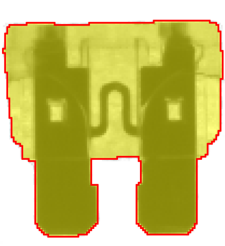
Figure 2.6: Part of the contour of an example region.

Such points may be computed very efficiently in RLE region representation. Let us observe that a contour of a region is defined by the sequence of its one pixel long, directed, **vertical segments** - once we have the vertical segments, adding horizontal sections between them is straightforward.

Moreover, all such vertical segments are easy to obtain from the RLE representation - these are in fact two sides of every point run in the region. Extraction of the contour therefore boils down to assigning successor to each vertical segment, i.e. answering a question for each such segment: which vertical segment will be next on the contour? This can be done one row of runs at a time with a careful linear scan of the point runs in the neighboring rows.

Calculating Region Features from Path Features

Contour extraction is a particularly useful feature, because it essentially converts pixel-precise region to subpixel-precise polygons; thus allowing convenient implementation of a number of region features that are more natural to define for polygons.



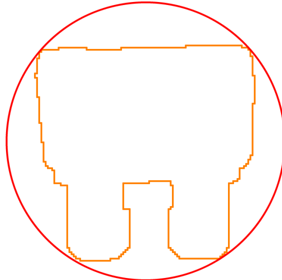


Figure 2.7: Bounding circle of a region computed indirectly as a bounding circle of its contour path.

Other Features

Other geometric features are not inherently related to pixel-precise regions, and either may be computed using contour extraction together with corresponding operator for sub-pixel precise shapes (e.g. bounding rectangle) or are

built on top of such operator (e.g. rectangularity factor). We will discuss these features in the **Contour Analysis** chapter.

Feature	**Description**
Bounding circle	The smallest circle containing the entire region.
Bounding rectangle	The smallest rectangle (of any orientation) containing the entire region.
Circularity	Measure of similarity to a circle.
Diameter	The longest segment between any two pixels of the region.
Perimeter length	Length of the region contours.
Rectangularity	Measure of similarity to rectangle.

Table 2.15: Other geometrical properties of a region.

> The following **Adaptive Vision Studio 4** filters extract the geometrical features of a region: `RegionBoundingBox`, `RegionBoundingCircle`, `RegionBoundingRectangle`, `RegionCircularity`, `RegionContours`, `RegionConvexHull`, `RegionConvexity`, `RegionDiameter`, `RegionPerimeterLength` and `RegionProjection` and `RegionRectangularity`.

2.7 Examples

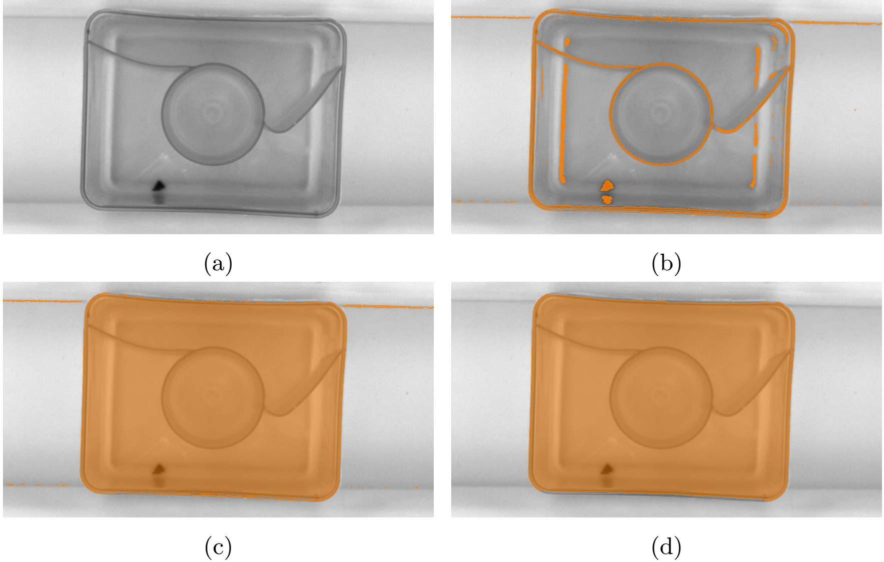
In this section we will present a couple of industrial problems solved using techniques introduced in this chapter.

Capsule Extraction

Region-processing techniques are commonly applied to refine inaccurate results of image thresholding. **Figure 2.8** demonstrates a process of acquiring correct representation of semi-transparent dishwasher powder capsule.

Transparency of the capsule material makes the object appear at similar brightness levels as the background, thus precluding application of global thresholding. Dynamic thresholding is applied instead to extract the boundaries of the capsule, along with unwanted horizontal edges of the conveyor line.

As long as we may rely on the capsule to have a closed, dark contour (which we assume we can), we can simply fill the region holes to acquire filled hull of the capsule and then perform morphological opening to remove unwanted thin traces of conveyor line. This process is illustrated in **Figure 2.8**.



(a) (b) (c) (d)

Figure 2.8: Input image (a), results of dynamic thresholding (b), filled holes of the extracted region (c), morphological opening applied to filled region (d).

Counting

In this example our aim is to count the teeth of a saw blade. Contrary to the previous example, now the region representing the object being inspected is flawlessly extracted using simple thresholding while the interesting part lies in the counting itself, i.e. in analysis of the extracted region.

Figure 2.9 demonstrates a morphology-based approach. Opening of the extracted region with big circular kernel removes the teeth, so that we may extract the saw teeth using region difference and connected components operators.

In industrial setting it would be prudent to perform slight dilation on the opened region before subtraction to make sure that none of the neighboring teeth pair will remain connected after that.

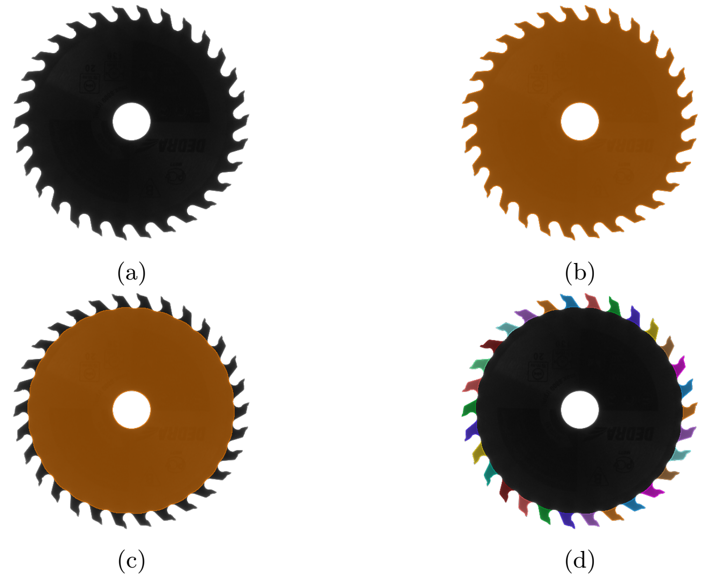


Figure 2.9: Input image (a), results of global thresholding (b), opening of the extracted region (c), connected components of difference between b. and c. (d).

Chapter 3

1D Edge Detection

The edge... There is no honest way to explain it because the only people who really know where it is are the ones who have gone over.

Hunter S. Thompson

3.1 Introduction

Image edges are locations of sharp change of brightness, i.e. locations of high local contrast. As a consequence of being a contrast-based feature, the presence and position of an edge is not altered by global illumination changes in the image; which contributes to the robustness of Edge Detection-based solutions.

Edge detection techniques come in two variants depicted in **Figure 3.1**. **1D Edge Detection** methods scan the image along a path and locate the points of intersection between image edges and the scan line. **2D Edge Detection** methods locate the entire edge. In this chapter we will inspect the first technique, featuring remarkable performance and sub-pixel precision.

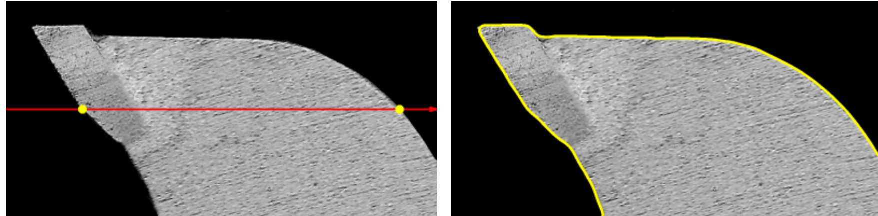


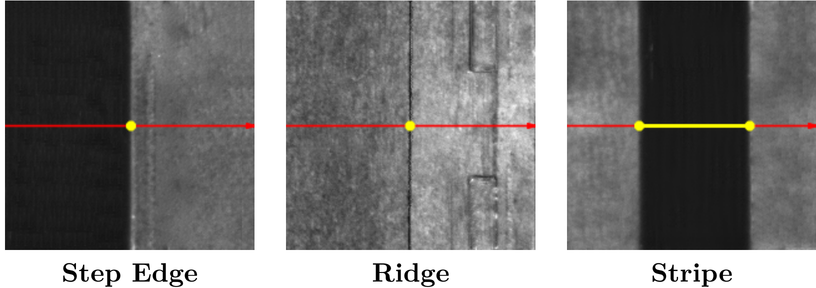
Figure 3.1: **1D Edge Detection**, **2D Edge Detection**

We will start with a short description of the methodology of extracting a 1D image brightness profile along a given path and then proceed to detection of the features present in the profile.

Depending on the nature of the brightness change that constitutes an edge, we distinguish two kinds of edges: **step edges**, occurring between two areas of different intensity, and **ridge edges** (or simply ridges), occurring where image intensity changes briefly and then returns to initial value.

A wide class of visual inspection tasks is focused on the areas bounded by two step edges of opposite polarity, rather than on the edges considered separately. Because of that it is useful to consider such areas as a third, additional type of feature discernible in one-dimensional profile; here called a **stripe**.

Overall, the chapter will cover detection of three kinds of features discernible in 1D profiles, all demonstrated in **Table 3.1**. In each example the feature (step edge, ridge or stripe) is vertical and the image is scanned horizontally to find the points of intersection between the feature and the scan line.



Step Edge	**Ridge**	**Stripe**

Table 3.1: Different kinds of image features extracted from 1D profile.

3.2 Profile Extraction

Before we apply any of the **1D Edge Detection** methods, firstly we need to acquire a 1D profile that is to be inspected. Computing a discrete profile of image brightness along a given path is a relatively straightforward task.

The first step is to sample the scan path, selecting a set of equidistant (typically with distance of one pixel [8, p. 150]) points of interest along the path. Each of these points will correspond to one element of the constructed profile. The second step is to compute the brightness values *related to* each of the points.

Multiple Sampling

We used the expression *related to* rather than *at* on purpose - although we could simply take the image brightness values at each point of interest, it is more prudent to use an average of a series of sampling points perpendicular to the scan line, as demonstrated in **Figure 3.2**; thus achieving a simple means of noise suppression.

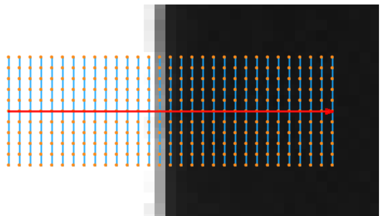


Figure 3.2: Multiple Sampling for 1D Edge Detection.

But what kind of average should we use to compute the result for a single point of interest? As long as the whole range of sampling points fits within the object being inspected and its features are perpendicular to the scan path, the brightness information collected at each of the sampling points is equally good or bad as the information collected at the other ones. Because of that we may safely use the simplest arithmetic mean.

We will refer to the number of points used to compute a single profile value as *scan width*. The wider the scan, the stronger noise attenuation we get. However, if the 2D feature we are to inspect is not perfectly perpendicular to the scan line, the wide scan area will cause the edge in the resulting 1D profile to be stretched and thus harder to identify precisely. Increasing the scan width will also increase the computation time of the profile extraction, which depends linearly on the number of sampling points.

Figure 3.3 demonstrates an example brightness profile extracted from the image on the right, scanned horizontally along the red scan line.

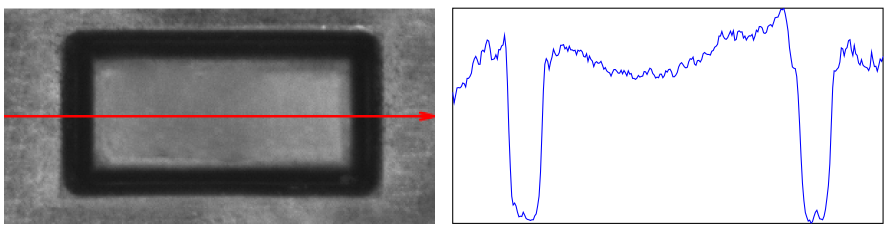


Figure 3.3: 1D brightness profile extracted from the image.

Refinement

Even though a reasonably wide scan area suppresses the noise significantly at the extraction level, we need to keep in mind that only random noise can be suppressed in this way. Real pictures contain small features[1] of image texture irrelevant to the inspection task that need to be attenuated before further processing of the profile.

Selecting the smoothing filter to refine the extracted profile is not an obvious choice. On the one hand we want to suppress the noise present in the profile, so that irrelevant intensity changes are not identified as edge points, on the other hand we want to achieve high precision of the edge localization.

These two criteria cannot be considered independently - smoothing of the profile suppresses the noise, but also lowers the precision. Canny determined[9] that the optimal trade-off between noise reduction and lose of precision is achieved with the Gaussian smoothing filter defined as follows:

$$g_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

where the standard deviation σ is a parameter of the filter.

Discreet Gaussian Filter

The Gaussian function is defined in continuous, infinite domain - to obtain a discreet approximation of the filter, we may sample $g_\sigma(x)$ at integer coordinates[10]. Moreover, as the value of the Gaussian function quickly decreases with the increase of $|x|$, we may limit the discreet filter to a finite neighborhood of $x = 0$ without significant effect on the results of the smoothing.

The well known fact called a three-sigma rule states that more that 97% of the Gaussian function integral is concentrated within 3σ distance from $x = 0$. It is therefore reasonable to sample the Gaussian function at $2r + 1$ points, where r is a small multiple of σ, e.g. $3\lceil\sigma\rceil$; thus obtaining a mask in the following form:

[1] Possibly perpendicular to the scan line and thus not affected by averaging the sampling points.

$$\frac{1}{s}\left[g(r) \quad g(r-1) \quad \ldots \quad g(0) \quad \ldots \quad g(r-1) \quad g(r)\right]$$

where s equals the sum of the obtained gaussian coefficients, i.e. it ensures the mean-preservation property of the filter.

Standard Deviation

Accurate adjustment of σ will contribute to the robustness of the computation. We should pick a value that is high enough to eliminate noise that could introduce irrelevant extrema to the profile derivative, but low enough to preserve the actual edges we are to detect. This parameter should be adjusted through interactive experimentation on representative sample data, looking for optimal trade-off between fidelity and noise attenuation.

Figure 3.4 demonstrates effects of smoothing the example brightness profile with different σ values. Blue profile ($\sigma = 0.0$) exhibits fine noise while brown profile ($\sigma = 6.0$) attenuates the valleys of significant edges, which makes both suboptimal. Red profile ($\sigma = 3.0$) seems to exhibit appropriate degree of smoothing.

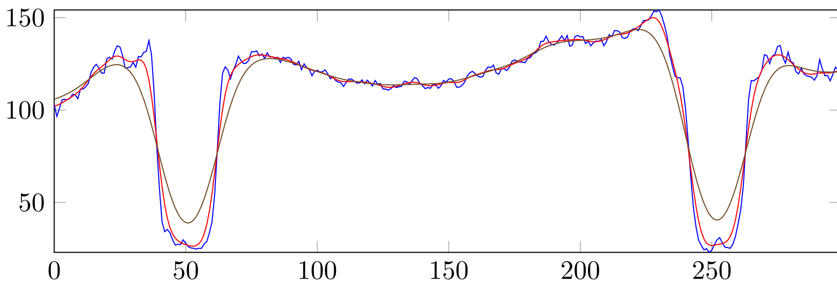


Figure 3.4: Original brightness profile smoothed with different standard deviations of Gaussian operator.

3.3 Step Edges

Once the brightness profile is extracted and smoothed, we can proceed to detection of its features. First type that we will inspect is **step edge**. Step

edges occur between two areas of different intensity and are represented as an abrupt intensity change in the 1D profile.

Edge Operator

Finding the step edges in a profile requires an edge operator - an operator that produces high output for locations representing sharp change of brightness and low output for the signal plateaus. One such operator is the **derivative** of a function - an elementary concept of calculus. This is also the operator suggested by Canny in already mentioned work [9]. But how do we actually compute the derivative?

In case of a continuous signal its derivative is well defined. As both image and (consequently) its brightness profile are discreet, we are left with partial differences - discreet approximations of the signal first derivative.

The simplest way to compute the partial difference of a discreet signal S is to subtract each value from its successor, i.e.:

$$D[i] = S[i+1] - S[i]$$

This operator, called **Forward Difference**, has a slight drawback - the resulting approximation $D[i]$ actually corresponds to the domain value in between $i+1$ and i (i.e to $i+\frac{1}{2}$). To achieve *stable* approximation of the first derivative we can compute the value $D'[i]$ as a mean of $D[i]$ and $D[i-1]$ (**Central Difference**):

$$\begin{aligned} D'[i] &= \frac{1}{2}(D[i] + D[i-1]) \\ &= \frac{1}{2}(S[i+1] - S[i] + S[i] - S[i-1]) \\ &= \frac{1}{2}(S[i+1] - S[i-1]) \end{aligned}$$

Both equations are feasible for our application, however we need to remember about the $\frac{1}{2}px$ shift introduced by Forward Difference operator and translate the edge points accordingly on the very end. **Figure 3.5** demonstrates an example finite difference profile.

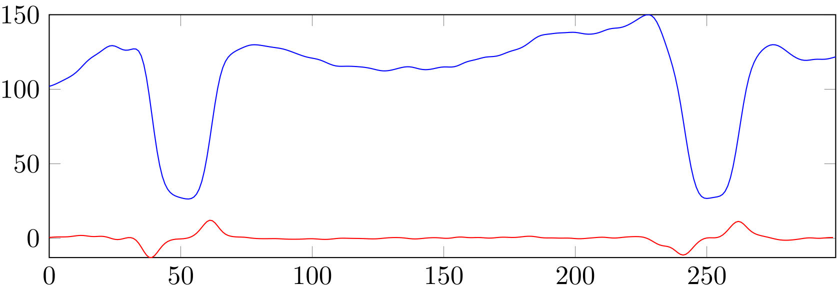


Figure 3.5: An example profile and its forward difference.

Edge Points

Once we have computed the derivative we can identify the edge points of the original profile. There are two criteria that a profile value has to meet to be considered an edge point:

1. Significant magnitude, i.e. magnitude larger than some predefined threshold.

2. Locally maximal magnitude.

Both conditions are necessary - first ensures that only significant brightness changes are identified as edge points, second (called Non-Maximum Suppression) ensures that a significant but stretched edge yields only one edge point.

The value of the minimum magnitude threshold in each case should be adjusted after inspection of derivative profile of sample data. Example depicted in **Figure 3.5** exhibits four significant peaks of the derivative profile varying in magnitude from 11 to 13, while the magnitude of its other extrema is lower than 3. In such case a value in the middle of range $(4, 10)$ would be a prudent choice of minimum magnitude threshold.

Profile locations meeting the magnitude criteria directly translate to edge points in the original image. An example set of extracted edge points is demonstrated in **Figure 3.6**.

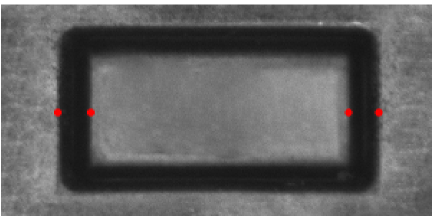


Figure 3.6: Result of **1D Edge Detection** - a list of edge points along the scan path.

Sub-pixel precision

Even though both the image being inspected and the extracted brightness profile are discreet (with pixel-precision), we can compute the local extrema of the derivative profile with sub-pixel precision thus achieving sub-pixel precision of the entire method.

Given a local extremum of a profile P at location i we can fit a parabola through three consecutive profile values: $P[i-1]$, $P[i]$, $P[i+1]$ and use the x-coordinate of its peak as the location of the extremum.

Edge polarity filtering

It is often useful to filter the extracted edge points depending on the transition they represent - that is, depending on whether the intensity changes from bright to dark, or from bright to dark.

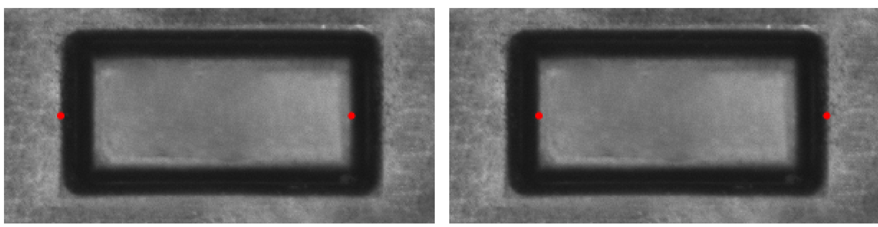


Figure 3.7: **inTransition** = BrightToDark, **inTransition** = DarkToBright

Post-processing

Once we have extracted the list of relevant edge points in the image we are nearly done. Depending on the use scenario, it may be useful to perform additional filtering of the extracted points on the very end of computation. Three methods of post-processing are particularly useful.

All Edges

One trivial post-processing method is to simply return all of the extracted step edges, that is - not to perform any post-processing at all. That would be the default method to follow whenever we want to detect the number of edges present in the image.

N Edges

Another option would be to select a fixed number of strongest edge points. If we know the number of edges in advance, this method allows us to disregard the adjustment of minimum magnitude threshold - we can simply set it to zero and expect that the actual edges will still be correctly located.

That being said, it is often useful to adjust minimum magnitude threshold anyway, so that in case of an error such as the object not being present in the image, the computation will explicitly fail instead of returning irrelevant weak edges.

Step edge detection algorithms are implemented in three **Adaptive Vision Studio 4** filters. All of them share common extraction logic, differing only in post-processing method applied to select the final outcome.

- `ScanMultipleEdges` - returns all of the extracted edges.
- `ScanExactlyNEdges` - selects the most prominent set of edges of given cardinality.
- `ScanSingleEdge` - wrapper over previous filter which selects the single most prominent edge.

3.4 Ridges

Ridges are brief bright or dark impulses on a contrasting background. Differing from step edges in their definition, they also require slightly different method of extraction. We will start the description from the point in which we have just extracted and refined the 1D profile of image brightness, as illustrated in **Figure 3.8**.

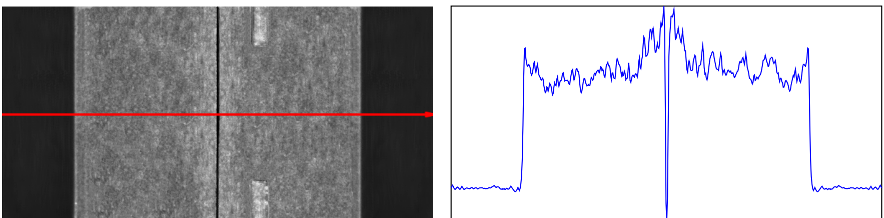


Figure 3.8: 1D brightness profile of an image with strong ridge in its center.

Ridge Operator

Ridges can be thought of as pairs of step edges of opposite polarity lying extremely close to each other. We could use this observation to propose a simple ridge detector operator adding together results of Forward Difference and Backward Difference operators:

$$\begin{aligned} R[i] &= (S[i] - S[i-1]) + (S[i] - S[i+1]) \\ &= 2S[i] - S[i-1] - S[i+1] \end{aligned}$$

Such operator would be a discreet equivalent of the ridge operator proposed by Canny[9], it has however two important drawbacks, pointed out by Subirana-Vilanova and Sung[11]:

- The operator has non-zero response for step edges, which can easily lead to false-positive errors.
- The quality of the detection strongly depends on the ridges having exactly 1 pixel in width, while in reality ridges usually appear as at least slightly wider.

Both problems are illustrated by **Figure 3.9 -** we can notice a high impulse response for step edges on the boundary of the object. Moreover, as the ridge in the original image has three pixels in width, it appears in the resulting profile as a pair of consecutive step edges.

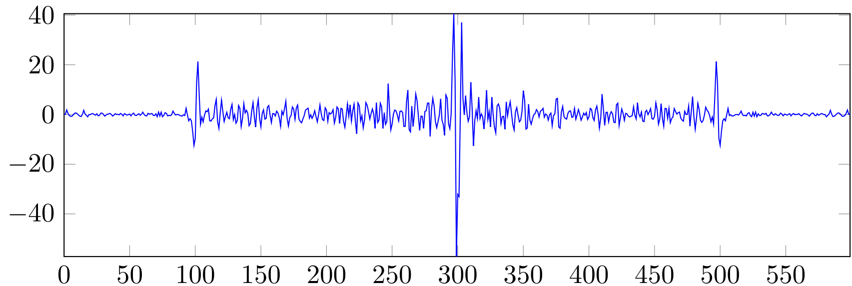


Figure 3.9: Naive ridge detection operator applied to the example profile.

The authors of [11] suggest to solve the problem of high response to step edges by applying each half of the ridge operator separately and using the minimum of two responses.

$$R[i] = \min(S[i] - S[i-1], S[i] - S[i+1])$$

Such form of the operator is already feasible for narrow (one pixel wide) ridges. To successfully detect wider ridges we could define a general operator parametrized by the width of the ridge and the width of the reference margin as follows:

$$\begin{aligned} R[i] \quad = \min(& \\ & \overline{S[i..(i+Width)]} - \overline{S[(i-1-Margin)..(i-1)]}, \\ & \overline{S[i..(i+Width)]} - \overline{S[(i+1)..(i+1+Margin)]}) \\ ) & \end{aligned}$$

where $\overline{S[a..b]}$ denotes the average of S values between a-th and b-th element, both inclusive.

It should be noted that contrary to the edge detection operator which could be applied regardless of the polarity of edges being extracted, our ridge detection operator (because of the minimum function) works specifically for bright ridges. To extract dark ridges, analogous equation with maximum operator should be used.

Figure 3.10 demonstrates the outcome of using such operator on the example data from **Figure 3.8**. The maximum operator suppresses the magnitude of negative values (indicating possible ridge candidates) but amplifies the magnitude of positive values. For clarity the drawing was cropped to negative-y part.

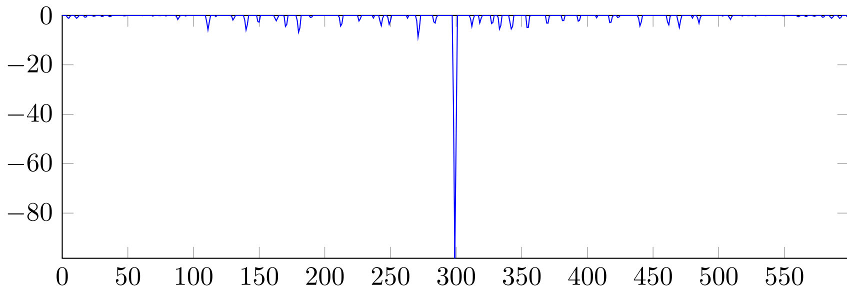


Figure 3.10: Amended ridge detection operator applied to the example profile.

Example results of ridge detection performed using such operator are demonstrated in **Figure 3.11**.

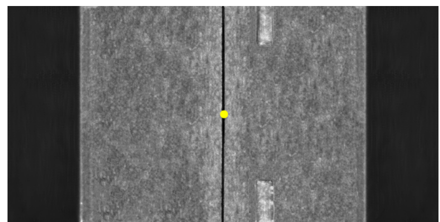


Figure 3.11: Results of ridge detection.

Post-processing

All methods of post-processing of the extracted step edge points described in Step Edges section are applicable to ridges.

> Ridge detection algorithms are implemented in three **Adaptive Vision Studio 4** filters. All of them share common extraction logic, differing only in post-processing method applied to select the final outcome.
>
> - `ScanMultipleRidges` - returns all of the extracted ridges.
> - `ScanExactlyNRidges` - selects the most prominent set of ridges of given cardinality.
> - `ScanSingleRidge` - wrapper over previous filter which selects the single most prominent ridge.

3.5 Stripes

Stripes are flat sections of brightness profile bounded by two step edges of opposite polarity. Such definition indicates that the problem of stripe detection heavily depends on the already discussed detection of step edges.

The concept of stripe is important mostly as a clear and succinct means of formulation for a range of visual inspection tasks, whereas it does not bring any novelties to the signal-processing aspect of the computation. Indeed, algorithms for stripe extraction firstly find the step edges in the profile (using previously described methods) and then process the results combining the extracted edges into stripes.

Next section summarizes two basic methods of combining the extracted step edges into stripes.

Edge Processing

All Stripes

As long as our goal is to maximize the number of constructed stripes, the problem can be solved quite efficiently. It can be easily proven that a simple

$O(n)$ algorithm that greedily connects each closing edge with the first opening edge between already constructed stripes and the closing edge itself yields optimal results.

N Stripes

The task is slightly more complicated if we know the desired number of stripes in advance and aim at maximizing the sum of strengths of step edges constituting the selected stripes. To solve such optimization problem in $O(n^2)$ time we can use a dynamic programming solution.

Let us define a partial solution to the problem as follows:

`Best[Prefix][Count]` - sublist of the first `Prefix` step edges of alternating edge-polarities having `Count` elements that yields the biggest sum of edge strengths.

Having computed the results for `Prefix` $= p$ we can compute the results for `Prefix` $= p + 1$ in linear time - for each `Count` $= c$ we need to consider only two cases, either using the $p + 1$-th step edge to extend the optimal result of `Best[p][count-1]` or not, in which case the result for the subproblem will be equal to `Best[p][count]`.

Stripe detection algorithms are implemented in three **Adaptive Vision Studio 4** filters. All of them share common extraction logic, differing only in post-processing method applied to select the final outcome.

- `ScanMultipleStripes` - maximizes the number of returned stripes.
- `ScanExactlyNStripes` - constructs the most prominent set of stripes of given cardinality.
- `ScanSingleStripe` - wrapper over previous filter which selects the single most prominent stripe.

3.6 Examples

In this section we will demonstrate a few industrial applications of 1D Edge Detection methods.

Positioning

1D Edge Detection methods are commonly employed to determine locations of objects. Let us consider an image of a capsule on a production line demonstrated in **Figure 3.12**. We assume that the capsule is aligned with the axis of the image and we want to determine the range of x-coordinates occupied by the object.

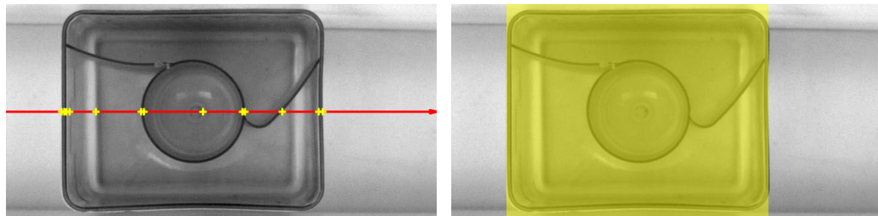


Figure 3.12: 1D Edge Detection applied to determine capsule position along the x axis.

As long as the background is plain and contrasting with object border, such problem can be solved easily regardless of the inner content of the object. **Figure 3.12** demonstrates the edge points detected along the horizontal scan line and visualisation of the resulting capsule position. The algorithm detects redundant, inner edges, but this does not pose a difficulty, as we only need the first and the last edge point of the returned list.

Code Reading

One of the classic applications of stripe detection is reading of 1D data codes. Depending on the barcode format, we may[2] or may not[3] know the number of bars the code is composed of - in the first case we may improve the robustness of the method using the post-processing routine for extracting the fixed number of strongest stripes which we have discussed before. In either case we expect the method to measure the width of the bars present in the image.

[2]E.g. in case of codes from EAN/UPC family commonly used in trade.
[3]E.g. in case of Pharmacode or Code128 standard.

Figure 3.13: 1D Edge Detection applied to read the widths of 1D code bars.

It is interesting to note that the intercept theorem guarantees that we may scan the barcode at any orientation of the scan line, as long as its deviation from the barcode axis does not disrupt the stripe detection routine - the proportions of widths of the intersected bars are preserved under any orientation of the scan line.

Once the widths of the bars are obtained they can be passed to a decoder for the specific barcode format to obtain the final reading.

Chapter

4

2D Edge Detection

So no, I don't think we've lost our edge at all.

Vince McMahon

4.1 Introduction

In the **1D Edge Detection** chapter we have discussed a set of techniques designed to detect high-contrast features in one-dimensional profiles. Let us recall that doing so we were not interested in one-dimensional image profiles by themselves, but in two-dimensional image features intersecting the line from which the profile was extracted: we have been transferring the extracted edge points back to the original image, acquiring fragments of information about the edges present in the image.

As we know how to detect individual edge points in an image, we may wonder if we could use the same techniques to detect entire edges - repeatedly detecting edge points and connecting them to form two-dimensional objects. The answer to this question depends on the context of the detection.

We cannot do that when we have no information about the edges that we are going to detect, as **1D Edge Detection** techniques relay on the position and orientation of the scan line, which has to be roughly perpendicular to the primitive being inspected. However, when we do have an estimate of the feature being extracted, such approach is valid and promising - we will discuss this idea further in the **Shape Fitting** chapter.

In this chapter we will look into the problem of **2D Edge Detection** from the ground up, incorporating the two-dimensional nature of the features into the detection algorithm, and obtain techniques that identify the edges in an image without any prior knowledge about them.

4.2 Image Gradient

A derivative of a signal is a measure of local change in the signal value, and as such is a natural starting point for detection of step edges, regardless of the dimension of the space in which the signal is defined. A derivative of two-dimensional signal is usually called a **gradient**.

In the **1D Edge Detection** chapter we have been using partial difference operators precessed by smoothing of the signal to obtain the derivative of a profile and identify its prominent extrema as the edge points. We will translate this approach to two-dimensional space preserving its core idea - calculation of signal derivative using smoothing and partial difference operators.

2D Partial Difference

To calculate the gradient of an image we may apply one of the partial difference operators discussed in the **1D Edge Detection** chapter separately in X and Y dimension. **Figure 4.1** demonstrates appropriate convolution masks for the central difference operator.

$$D_H = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \qquad D_V = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

Figure 4.1: Horizontal and vertical central difference masks.

In effect we obtain two images (or one two-channel image), representing vertical and horizontal derivatives at each pixel. Combining these two components into vectors representing two-dimensional gradient yields a result demonstrated in **Figure 4.2**, where the resulting gradient vectors were drawn onto the original image.

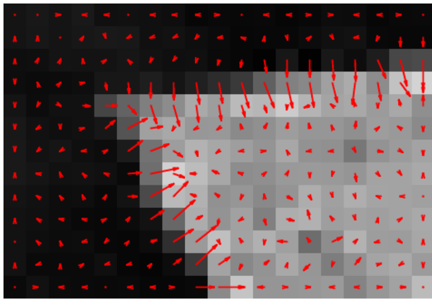


Figure 4.2: Two-dimensional image derivative computed at each pixel.

For our needs of edge detection, we are interested mainly in the magnitude of gradient vectors. **Figure 4.3** demonstrates a gradient magnitude image, each pixel of which represents the magnitude of the corresponding gradient vector (computed using central difference operators as described above).

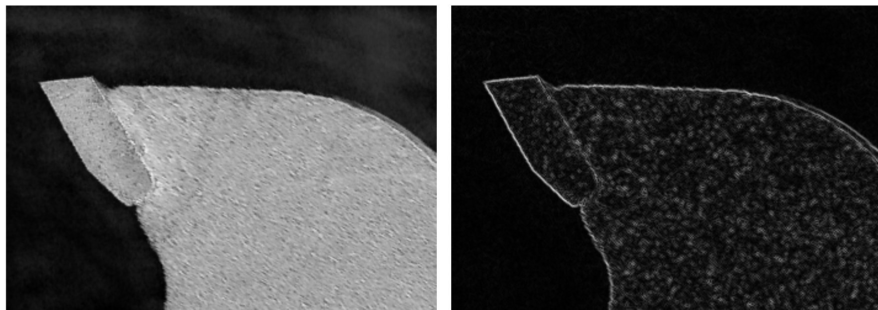


Figure 4.3: Gradient magnitude image of the example image.

We may notice that image edges are indeed discernible in such image, even though there is also significant amount of high-gradient pixels corresponding to fine features of the saw texture, rather than object edges. Sensitivity to noise was the reason for which we have incorporated Gaussian smoothing operator into our one-dimensional edge detection routines; it is clear that we need similar precautions in two dimensions.

The need for simple operators

Even though the Canny's result[9] about the optimality of the derivative of Gaussian operator for edge detection applications holds for 2D, we will commence out description with simpler gradient operators, moving to Canny's method in the next section.

Contrary to the case of one dimension, in two dimensions the amount of data to be processed is usually quite large, which creates a trade-off between accuracy and speed for real-time high-demand inspection systems. It is therefore reasonable to look into simple and fast operators, even if they are less accurate that the one advocated by Canny.

Prewitt Operator

One of the first gradient operators, Prewitt operator, addresses this issue by means similar to the Multiple Sampling technique which we employed to suppress noise in profiles of image brightness at the extraction level.

Here the central difference operator is applied to three rows/columns in the immediate neighborhood of each pixel, which is equivalent to convolving the central difference operator witch mean average operator running in the opposite direction, as demonstrated in **Figure 4.4**.

$$P_H = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

$$P_V = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 4.4: Masks of the Prewitt gradient operator.

Sobel Operator

The quality of noise-suppression of the Prewitt operator may be improved if we replace the indiscriminate mean average with simple approximation of Gaussian smoothing, which puts emphasis on pixels nearer to the pixel at which the average is computed. Such operator was proposed by Sobel.

$$S_H = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_V = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 4.5: Masks of the Sobel gradient operator.

Nixon and Aguado provide[12] an interesting description of derivation of similar masks for higher dimensions, where both smoothing and differencing components are obtained from Pascal's triangle, however such masks are rarely used in practice. Sobel masks of bigger dimensions approximate the Gaussian smoothing while not making use of the separability of the Gaussian kernel, which make them inferior to the actual Gaussian smoothing, as performed by Canny edge detection method.

Figure 4.6 compares the results of bare central difference operator with those obtained using the Sobel operator. The achieved noise suppression is not spectacular, but still noticeable.

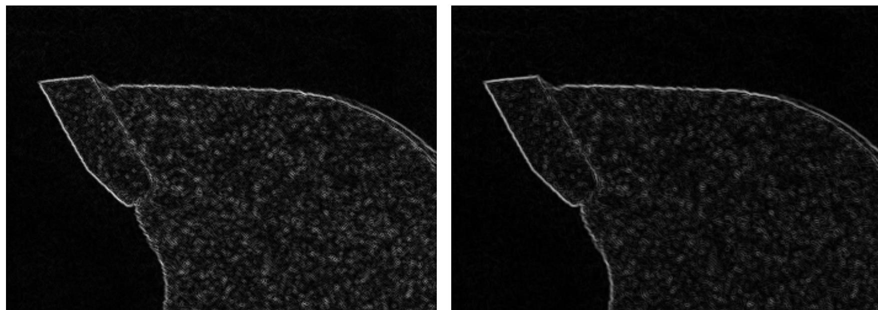


Figure 4.6: Comparison of bare central difference gradient (on the left) and Sobel operator gradient.

Scaling

When we derived the masks of Prewitt and Sobel operators we neglected the scaling of the smoothing operators, i.e. we used smoothing masks such as $[1\,1\,1]$, rather than $\frac{1}{3}[1\,1\,1]$. The operators were defined in this way to allow for faster computation and are commonly used in this form.

Whenever the smoothing operators are to be compared (or used interchangeably, e.g. in edge detection with fixed gradient magnitude threshold) it is important to remember to normalize their results, multiplying the outcome by the scaling factor of the smoothing operator omitted in the first place ($\frac{1}{3}$ for the Prewitt operator and $\frac{1}{4}$ for Sobel). Such scaling was applied to the results of the Sobel operator in **Figure 4.6**.

> Gradient operators discussed in this section are implemented by **Adaptive Vision Studio 4** filters `GradientImage_Mask` and `GradientImage`.

4.3 Canny Edge Detector

Having discussed the concept of image gradient and a set of simple fixed-mask gradient operators, now we move our attention to the complete step edge detection technique described[9] by Canny (the one-dimensional version of which was our method of choice for detection of step edges in image profiles).

Let us recall that Canny's method for one dimension may be summarized as follows:

1. Compute the derivative of Gaussian of the signal.

2. Select the resulting points that:

   a) Have magnitude above certain minimum value (**Thresholding**)

   b) Have locally maximum magnitude (**Non-maximum Suppression**)

Canny's method for two-dimensional edges retains this schema, but some of 2D-specific details of the individual steps are interesting enough to call for a detailed discussion.

Gradient by derivative of Gaussian

The Gaussian function, which we have discussed in the **1D Edge Detection** chapter, has a natural formulation in two dimensions:

$$g(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Apart from the optimal[1] trade-off between noise attenuation and edge preservation, the two-dimensional Gaussian filter has two useful, 2D-specific features. The first one is **isotropy** - the response of the filter depends only on the distance from (x, y) to $(0, 0)$; thus the smoothing does not introduce any bias dependent on the direction of the image edges.

[1] Under the criteria assumed by Canny.

The second interesting feature is **separability**, which means that the application of the two-dimensional Gaussian filter is equivalent to successive application of two one-dimensional operators, each processing the signal in different dimension. Indeed:

$$\begin{aligned} g_\sigma(x,y) &= \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \cdot \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2}{2\sigma^2}} \\ &= g_\sigma(x) \cdot g_\sigma(y) \end{aligned}$$

Therefore to perform the Gaussian smoothing on an image, we may simply apply the already discussed one-dimensional smoothing operator successively to each row, and then to each column of the image. Gradient extraction based on such smoothing is demonstrated in **Figure 4.7**.

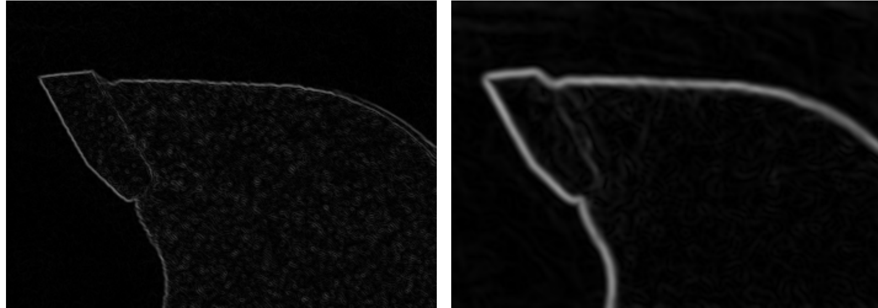


Figure 4.7: Gradient extraction using Gaussian smoothing with different values of standard deviation (0.7 on the left, 2.75 on the right).

The example results look promising - in the image on the right the outline of the object is prominent and consistently brighter that any other element of the image; the noise is effectively suppressed. On the other hand, the edges in the image clearly have excessive width, which inhibits the precision of edge localization. Conversely, image on the left exhibits fine edges, but also worse separation of edge and noise intensities. It is worth stressing that accurate adjustment of σ is crucial for performance of the algorithm.

Hysteresis Thresholding

Once we have computed the image gradient, we need to threshold the gradient magnitude image selecting the high-gradient, prospective edge locations. Although simple global thresholding described in the **Image Thresholding** chapter could be employed for that, Canny pointed out an important drawback of this method in the context of edge detection.

Let us assume that we set the minimum magnitude threshold value to s, and the image contains an edge mean magnitude of which is near to s. Even if the illumination of the scene is uniform, the gradient magnitude of the edge pixels is bound to fluctuate around s due to inevitable noise introduced by camera imperfections.

In practice the fluctuation of edge gradient magnitude tend to be quite significant, leading to extraction of fragmented, incomplete edges. **Figure 4.8** demonstrates two ineffective global thresholding attempts on example gradient amplitude image. The results on the left are free from noise but incomplete, while lower threshold used to produce the outcome on the right introduces significant amount of noise.

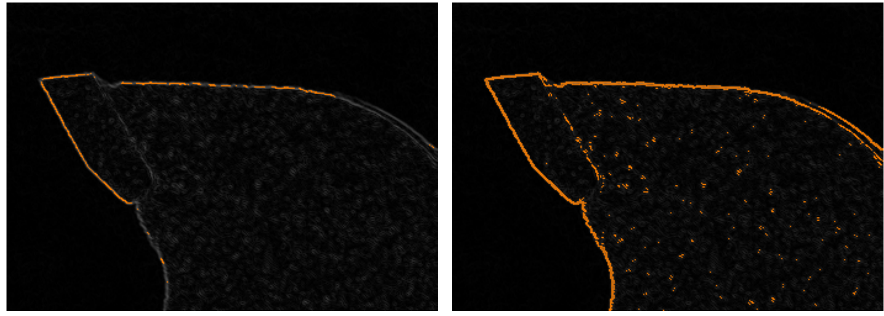


Figure 4.8: Global thresholding of the gradient magnitude image.

Canny proposed[2] to address this issue by setting two thresholds, let us denote them by l and h, where $l < h$. The locations of magnitude lower than l are immediately discarded, while locations of magnitude higher than h are immediately accepted. The locations of magnitude between l and h are split into connected components and each component is accepted only if it is adjacent to previously accepted (i.e. of magnitude higher than h) locations.

Such topological extension of the usual thresholding, called hysteresis thresholding, proves extremely useful in edge detection applications. **Figure 4.9** demonstrates hysteresis thresholding of the example image using two threshold values used to produce results in **Figure 4.8** as l and h. As we can see, the edges from the low threshold image are retained in their entirety, while all of the noise is attenuated.

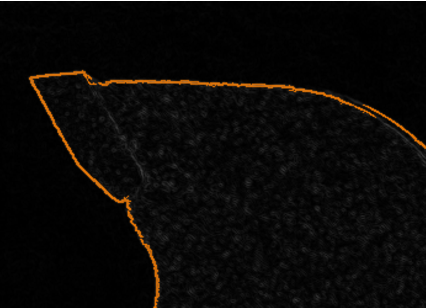


Figure 4.9: Hysteresis thresholding of the gradient magnitude image.

Non-maximum Suppression and Subpixel Precision

A direct translation of one-dimensional condition of locally maximal magnitude of edge locations would be to compare each pixel with its neighbors (either 4 or 8), but such approach would effectively yield only isolated, single edge locations and this is clearly not what we want.

[2]This approach is a two-dimensional equivalent of the method proposed[13] by Schmitt in the context of electronic trigger, although Canny did not indicate his inspirations on that matter.

To derive correct translation of non-maximum suppression into two dimensions we need to remember that in 1D edge detection we were analyzing profiles perpendicular to actual image edges. As the direction perpendicular to an edge is directly represented by gradient direction, non-maximum suppression in two dimensions should be performed by comparing each pixel magnitude with its two neighbors indicated by the pixel gradient direction.

We may use a similar approach to translate out parabola-fitting technique of acquiring sub-pixel precise positions of edge points to two-dimensions. If we sample the image at three points on the line parallel to gradient direction at a pixel and fit a parabola through the resulting, three-element profile, we obtain a direct equivalent of the fit that we were doing in one-dimensional case.

> Edge detection method is implemented by four **Adaptive Vision Studio 4** filters, which share common logic, differing only in output data type. One can choose between Canny, Deriche and Lanser filters, or between Sobel and Prewitt algorithms when using *_Mask* version of filters.
>
> - `DetectEdges_AsPaths` - returns extracted edges as array of paths.
> - `DetectEdges_AsRegion` - extracts edges and returns them as one region.
> - `DetectEdges_AsPaths_Mask` - works as DetectEdges_AsPaths, but can be used with Sobel or Prewitt filters for gradient computing.
> - `DetectEdges_AsRegion_Mask` - the same as previous filter, but result is stored in region rather than in array of paths.

CHAPTER

5 Contour Analysis

> Be precise. A lack of precision is dangerous when the margin of error is small.
>
> DONALD RUMSFELD

5.1 Introduction

Contour analysis can be thought of as a sub-pixel precise counterpart of **Blob Analysis**, built upon processing of precise points organized in paths; in the same way that **Blob Analysis** is based on processing of pixel-precise regions.

Solutions based on contour analysis typically follow the schema of **Blob Analysis** solutions:

1. **Extraction** - sub-pixel precise paths are extracted from the image, usually by means of **2D Edge Detection**.
2. **Processing** - the paths are subject to transformations with an intention of rectifying the path imperfections or extracting its segments that we are particularily interested in.
3. **Feature Extraction** - in the final part the numerical and geometrical features may be computed.

5.2 Path

Path, the fundamental data type of contour analysis, is a simple list of points defining two-dimensional curve consisting of straight segments; along with one additional, boolean information indicating if the list is **closed** or **open**, i.e., if the first point should be interpreted as connected with the last one.

Both types of paths occur naturally as results of **2D Edge Detection**, as demonstrated in **Figure 5.1**. Paths may be also extracted from regions by computing region contours (always closed paths) or medial axes (both open and closed).

Characteristic Points

Each non-degenerate path represents an infinite number of points on the segments between its defining points. To avoid confusion we will consequently refer to the defining points of a path as its **characteristic points**.

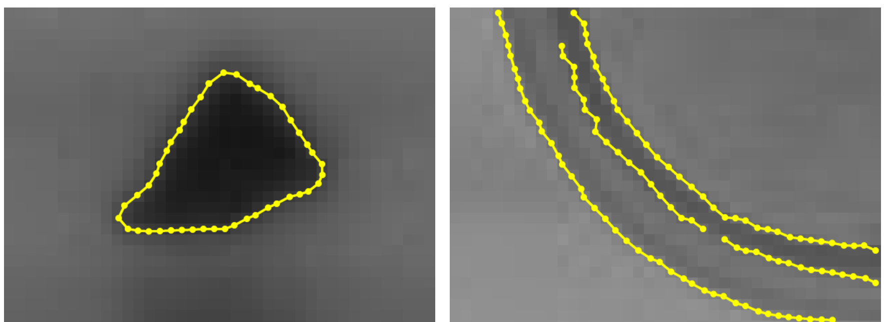


Figure 5.1: Closed and open paths extracted in **2D Edge Detection**.

Polygons

Let us note that our definition of path is quite general - we allow the path to have duplicate points and self-intersections. This is reasonable because a lot of important path operators (e.g. the segmentation routines which we are going to discuss soon) and features (e.g. path length) are well-defined for such paths and require no additional restrictions.

One exception is extraction of features defined in the context of two-dimensional shapes enclosed by a path, i.e. sub-pixel precise polygons. Such operators make sense only when the path divides the space into exactly two parts - interior and exterior, and thus has to be **closed** and **not self-intersecting**.

5.3 Segmentation

It is often the case that we are interested in a specific section of the obtained object contour, e.g. if we want to measure the angle between two specific straight segments of the object boundary. Segmentation techniques extract the sections of a path that represent simple geometric primitives such as straight segments or circular arcs.

The task of path segmentation may be also formulated in terms of path approximation - if we approximate a path with a simple polygon, we can split it into sections corresponding to individual segments of the resulting polygon, thus obtaining segmentation into straight segments.

The algorithm that will be our basis for developing the path segmentation methods was introduced as an algorithm for path polygonal approximation.

Ramer Algorithm

Ramer proposed[14] a simple and effective algorithm for approximating a path with a polygon within a given precision. The algorithm is parametrized with a value d, denoting maximal distance from the original curve to the resulting polygon.

Ramer algorithm constructs the approximating polygon from the points of the original path, i.e. vertices of the resulting polygon are a subset of the original path points. The algorithm commences by approximating the whole path with a single segment: for open paths it is the segment between the path endpoints, for closed paths we may select the endpoints a and b of a path diameter and process two open sections of the path (a to b, b to a) separately.

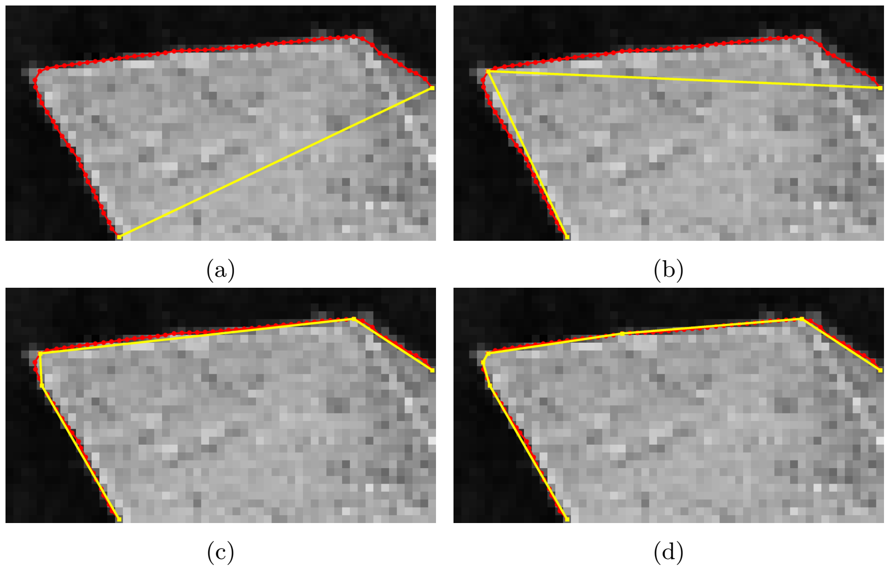
The algorithm proceeds recursively. At each recursive call the algorithm considers a section of the original path already approximated with a single segment S and finds the point P of the path section that is most distant to S. If the distance between P and S is smaller or equal to d, then the recursive call returns S as a feasible approximation for the path section. Otherwise the path is split at P and the algorithm is run recursively at each of two created sections.

Figure 5.2 demonstrates the partial results of Ramer algorithm at four levels of recursion, the last image demonstrating the final outcome.

Ramer algorithm may be also used as a means of lossy compression, although such application is rarely seen in practice, as the object contours are usually quite space-efficient, at least when compared to the size of the original images.

Straight Segments

Ramer algorithm may be directly applied to segment a path into straight sections. Once we have obtained the polygonal approximation, we need to split the path at the vertices of the polygon to produce an array of path sections, each of which will correspond to a segment that approximates it with the precision we have decided on.



(a) (b) (c) (d)

Figure 5.2: Ramer algorithm run on example path with $d = 1px$.

Results of this approach are demonstrated in **Figure 5.3**.

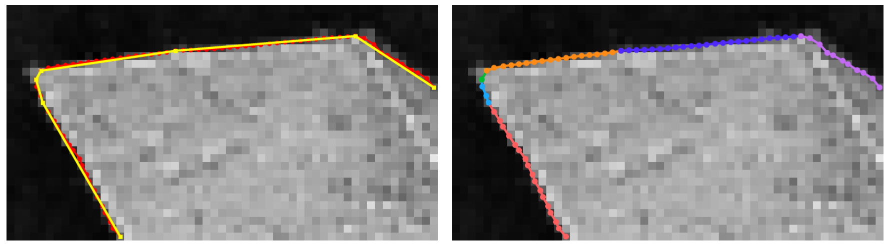


Figure 5.3: Segmentation of example path into straight segments based on the output of Ramer algorithm.

Circular Arcs

Another common type of feature that may be identified in object contours are circular arcs. A number of approach may be used to look for arciform sections

directly - one idea would be to analyze turn angles at consecutive characteristic points looking for runs of roughly constant curvature, as would be expected of a circular arc. Unfortunately, this approach does not perform well for big arcs - it is often the case that the turn angles at individual characteristic points are very weak and are easily affected by noise.

Another approach would be to adapt the Ramer algorithm to use arcs rather than straight segments. The modification is not straightforward, as two endpoints of a path section do not define a unique arc (as they did in case of segments), so we would need to fit an arc to each section by means of **Shape Fitting**. While this does not pose a big problem, we may note that the contours being inspected rarely consist solely of circular arcs[1]; usually they are a combination of straight and arciform sections.

To obtain a technique that would segment a path into sections of both types, we may start with the classic Ramer-based segmentation into straight sections and post-process its results, joining together short parts that would be well approximated by an arc.

One approach for such post-processing would be to fit an arc to each pair of consecutive sections of Ramer-segmented path and compute the maximum distance between the path and the arc, as well as the path and its current approximation[2]. If the maximum distance to the arc is lower, we may conclude that the arc is better approximation for the consecutive segments and join these together. This routine would be continued until convergence and in the end we would classify the sections that resulted in such joining as arciform and the others as straight.

Two problems may be encountered when this method is applied:

- When the arc being fit to a pair of consecutive straight sections has especially big radius and small length, so that it is close to a segment, the resulting improvement of the approximation is tiny and classification of the part as an arciform section may be confusing.

[1] Full circles are a common case, but these represent a single primitive and thus do not need to be segmented.

[2] Two straight segments or an arc in case of a part that is a result of such joining itself.

- Once two straight sections are joined, the resulting arc approximation may be so accurate that it will preclude further joining of the resulting section - a perfectly arciform path initially segmented into numerous straight parts has little chances of being fully joined.

First issue may be addressed by limiting the radii of the arcs being fitted to a predefined threshold (which will be a parameter of the algorithm) - if the resulting arc is to big we simply do not join the sections we are considering. This simple solution is perfectly feasible as long as the arcs we intend to extract have reasonably high relation of length to radius.

A natural solution to the second problem is indiscriminate joining of the consecutive sections whenever the maximum distance between the path section and the fitted arc is within the maximum distance we used for the initial Ramer segmentation.

Figure 5.4 demonstrates an example application of Ramer segmentation with post-processing with the amendments described above.

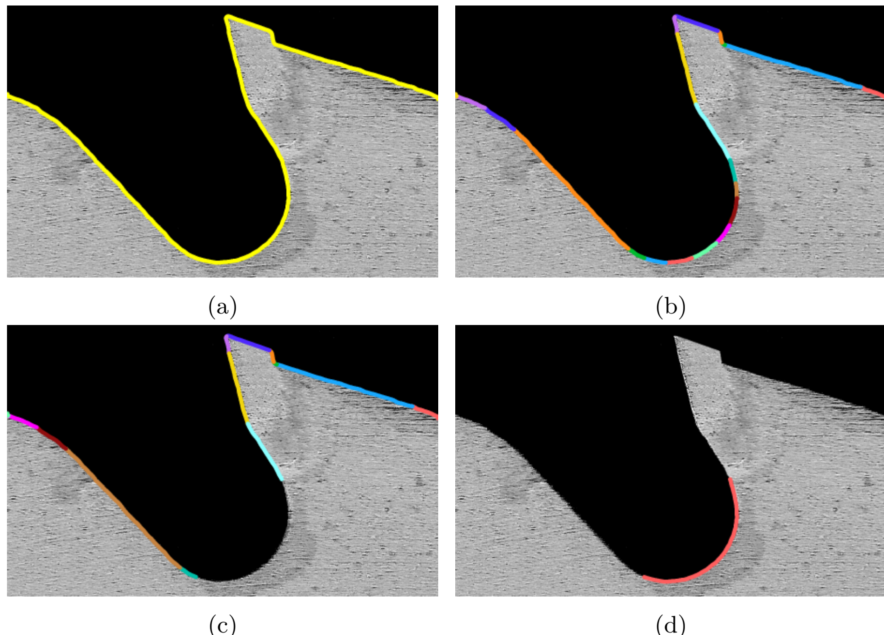
> **Adaptive Vision Studio 4** filter `ReducePath` implements the Ramer algorithm, while `SegmentPath` implements the segmentation technique built upon it as described above.

5.4 Statistical Features

Polygon Moments

In the **Blob Analysis** chapter we have discussed region moments - generalized statistical features based on region pixel coordinates. Equivalent formulas may be formulated for sub-pixel precise polygons - yet we will need to perform an integration over the polygon surface rather than simple summation, as the number of points contained in a non-degenerate polygon is infinite.

In the following equations we assume S to be a polygon defined by a closed, not self-intersecting path. The raw and central moments of S are defined as:



(a) (b) (c) (d)

Figure 5.4: Initial segmentation of the example contour (a) into straight segments (b) and results of post-processing: straight (c) and arciform (d) path segments.

$$m_{p,q} = \iint_{(x,y)\in S} x^p y^q dxdy$$
$$c_{p,q} = \iint_{(x,y)\in S} (x-\overline{x})^p (y-\overline{y})^q dxdy$$

and both may be normalized, i.e. divided by the area a of S:

$$m'_{p,q} = \frac{1}{a} m_{p,q}$$
$$c'_{p,q} = \frac{1}{a} c_{p,q}$$

Computation

As opposed to region moments, the formulas for shape moments do not give us a direct algorithm for computing any of them - in each case we are left with a double integration to perform.

Luckily, Green's theorem defines the relation between any double integral over simple polygon and line integral over its boundary, which allow to substitute double integral by a sum of line integrals, each of which will be evaluated over a segment of the boundary path. For the line integrals of low order, closed formulas may be obtained, as given[15] by Turkowski for first and second-order moments.

For instance, this approach allows to obtain the following formula for zeroth moment of a shape defined by a closed, n-point path S:

$$m_{0,0} = \frac{1}{2}\sum_{p=0}^{n-1}(S[p]_x \cdot S[p+1]_y - S[p+1]_x \cdot S[p]_y)$$

which is the well known formula for the area of polygon based on the cross product of vectors from $(0, 0)$ to its consecutive vertices.

From the practical point of view it is important that the polygon moments can be computed precisely, deterministically in linear time (in terms of the number of their characteristic points).

Applications

As the order of moments (defined as a sum of the moment exponents, i.e. $p+q$ for $m_{p,q}$) increases, the shape-describing information carried by them gets harder for interpretation and conscious use. In the field of industrial inspection usage of moments is usually focused on the moments of order 0, 1 and 2.

0th order

There is only one zeroth moment - $m_{0,0}$, which equals the **area** of the shape. Having both exponents equal to zero, this moment indeed ignores the values of both coordinates of the points contained by the polygon. Apart from direct applications, this moment is frequently evaluated to normalize (i.e. divide by the area) the other, higher-order shape moments.

1st order

First order moments are the first moments the normalization of which yields useful information. The moments $m'_{1,0}$ and $m'_{0,1}$ represent, respectively, the average x and y point coordinates of the shape. Together they form the **mass center** of the shape - if we imagine the shape as a flat, rigid body, then it would remain in balance when positioned over a point support in its mass center.

The mass center allow us to define certain geometric features, such us the shape **radius**, which we will discuss in the next section.

2nd order

Second order moments are the first moments that reflect the relation between x and y point coordinates of the shape, and as such, are particularly interesting. Extracting practical information about a shape from its second order moments is not trivial though - to do so we will return to the mechanical analogy of flat rigid body, which we have just introduced.

Under such interpretation, the matrix of the second order central moments of the shape:

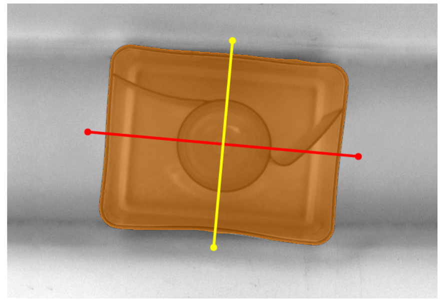
$$I = \begin{bmatrix} c'_{2,0} & c'_{1,1} \\ c'_{1,1} & c'_{0,2} \end{bmatrix}$$

represents an important concept of mechanics - **tensor of inertia**, which determines the torque needed to rotate a shape around any axis going through its mass center (in a similar way that a mass of a body determines the force needed to give the body the desired acceleration). The torque T needed to change the angular velocity of the body by ω may be calculated as follows:

$$T = I\omega$$

Although I determines the inertia for rotation around any axis going through the mass center, some of them are of particular interests - the principal axes ω', which yield stable rotation, i.e. preserve the direction of angular momentum. These axes represent the eigenvectors of I and as such may be obtained directly from shape moments.

Figure 5.5 demonstrates the principal axes of inertia of two example shapes.



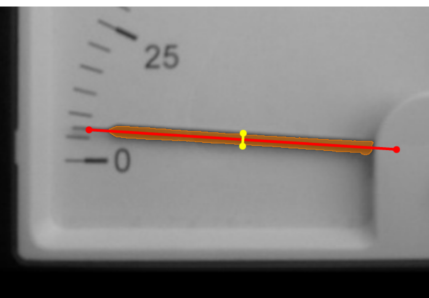


Figure 5.5: Principal axes of inertia (marked in red and yellow) of example shapes (marked in orange).

Interestingly, these axes may be also obtained using another, yet equivalent approach: as the axes of an ellipse having the same moments up to the second order as the shape being considered. An exhaustive description may be found in the textbook[16, p. 73-75] by Haralick and Shapiro.

The principal axes allow to define at least two useful shape features:

- **Orientation** - the direction of the major principal axis.
- **Elongation** - the quotient of major principal axis length and minor principal axis length.

> The following **Adaptive Vision Studio 4** filters extract the statistical features of a polygon: `ShapeArea`, `ShapeElongation`, `ShapeMassCenter`, `ShapeOrientation`, `ShapeEllipticAxes`.

5.5 Geometrical Features

Radius and Diameter

The principal axes of inertia which we have discussed recently provide approximate information about the dimensions of the shape. Strict, geometrical measures may be obtained as generalizations of basic features of circles: radius and diameter.

Radius of a shape may be defined as the maximum distance between its mass center and any of its points. It is not hard to prove that such point has to be a characteristic point of the polygon, and therefore may be found in a straightforward search.

Diameter of a shape may be defined as the maximum distance between any two points of the shape. Also in this case in may be shown that such points are always the characteristic points of the polygon and as such may be found in $O(n^2)$ time. This result may be improved to $O(n \log n)$ using the rotating calipers technique[17] on the convex hull of the polygon (see below).

Both these features offer an alternative definition for the **orientation** of a polygon - the vector from the mass center of the shape to the most distant point (corresponding to shape radius) is especially interesting, as it yields result in full range $(0°, 360°)$, as opposed to the moment-based orientation or the orientation of the diameter, which are limited to the $(0°, 180°)$ range.

Convex Hull and Convexity

Convex hull is the smallest convex polygon that contains a set of points. Finding such polygon is a classic problem of computational geometry for which numerous efficient algorithms has been developed. Popular textbook by Cormen et al. covers[18] some of them, probably the most popular one being the Graham algorithm.

Slightly less frequently mentioned method[19] by Andrew preserves the general idea of Graham algorithm yet uses lexicographic rather than angular sorting, which is simpler and less prone to errors (e.g. due to the imperfections of the floating-point representation) and as such it is worth considering for practical applications.

Once we know how to compute the convex hull of a path we may define a convexity factor of a shape as the quotient of the area of the shape and the area of its convex hull. The obtained numeric feature allows to estimate the magnitude of cavities and holes present in the objects being inspected.

Circle and Circularity

One trivial method to find the smallest circle containing a set of points is based on an observation, that such circle has to have some three of the given points on its boundary (or two in a special case when the points lie on the circle diameter) - otherwise we could shrink the circle and still cover all of the points.

We may therefore iterate over all triples of the given points, compute a unique circle passing through each of them and select the smallest feasible (i.e. covering all points) circle. Such solution would work in $\Theta(n^4)$ time, n denoting the number of given points, which may result in a significant computational burden even for relatively short paths.

Much faster solution based on iterative improvement was proposed[20] by Welzl - his randomized algorithm achieves linear expected running time and allows for a concise, recursive implementation.

We may define at least three reasonable numerical features that reflect the similarity between a given shape and a circle:

1. Quotient between the area of the shape and the area of its bounding circle.
2. Quotient between the area of the shape and the area of a circle with the same radius.
3. Quotient between the area of the shape and the area of a circle with the same perimeter.

All of these features assume values between 0 and 1, 1 being achieved for perfect circles. The selection of particular method should be based on the nature of expected deviation of the shape from a perfect circle. For instance, the second method, being based on a radius of the shape, will be particularly responsive to **elongated** shapes, while perimeter-based feature will be sensitive to high **curvature** of the object boundary.

> Some of the features being discussed in this section are well defined for all paths, including open and self-intersecting ones. These features

are implemented in **Adaptive Vision Studio 4** by filters from the Path Features category: `PathBoundingBox`, `PathBoundingCircle`, `PathBoundingRectangle`, `PathConvexHull`, `PathDiameter` and `PathLength`.

The following **Adaptive Vision Studio 4** filters extract the features defined only for polygons: `ShapeConvexity`, `ShapeCircularity` and `ShapeRectangularity`.

Chapter

6

Shape Fitting

Geometry was the first exciting course I remember.

Steven Chu

6.1 Introduction

Images being subject to interpretation in industrial inspection tasks often contain simple geometric shapes such as segments or circles. Precise extraction of such features is the key point for various measurement tasks, such as calculating the diameter of a cylinder head, demonstrated in **Figure 6.1**.

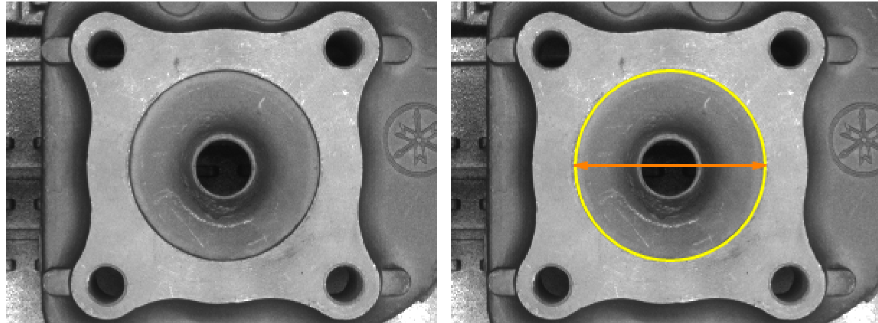


Figure 6.1: Example application of shape fitting - the diameter of a cylinder head is calculated indirectly as the diameter of a circle fitted to the image.

Various approaches may be taken to obtain abstract information about geometrical primitives represented in an image. The well known Hough transform performs an exhaustive search iterating through the space of all possible shapes, e.g. all possible radii and centers of circles, within a predefined constraints and with a predefined precision. Each candidate is verified using gradient direction information at the pixels that it is expected to intersect.

Unfortunately, such approach often proves computationally demanding and easy to disrupt by imperfections of the shape being identified. Moreover, the method requires careful calibration of the search space constraints and precision of the search. **Figure 6.2** demonstrates a typical problem that occurs frequently when the precision of the search is too low, causing the Hough transform to yield multiple results, none of which is accurate.

An alternative solution would be to apply one or two-dimensional **edge detection** to extract a group of edge points on the boundary of the shape being examined and fit the abstract shape to such points.

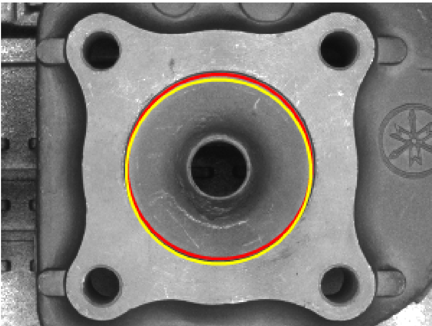


Figure 6.2: Hough transform-based circle detection performed on the example image.

In this chapter we will overview the classic methods of fitting lines and circles to sets of points in two-dimensional space. We will also demonstrate a particularily useful technique based on one-dimensional edge detection and shape fitting that allow to fit approximate positions of abstract primitives to their actual occurrences.

6.2 Lines

The problem of fitting lines to data frequently occurs in the field of statistics, where it is called *linear regression*. In this context the aim of the fitting is to find a line in the form $y = ax + b$ that optimally models the relation between the (controlled, error-free) variable x and (possibly affected by errors) y, given a set of data samples (x_i, y_i).

The popular slope-intercept representation of lines works well for statistical regression, where the x_i of data samples are unique, but causes problems in the general case, as the vertical lines can not be represented in this form and nearly-vertical lines require extremely high magnitude of the slope parameter, jeopardizing the numeric stability of the computation. We will therefore use the general equation of a line:

$$\{(x, y) : ax + by + c = 0\}$$

together with a normalization condition:

$$\sqrt{a^2 + b^2} = 1$$

The normalization of the vector (a, b) asserts that every line has a unique representation and inherently includes the necessary requirement of $(a, b) \neq (0, 0)$. It also allows easy computation of the distance between the line and any point (x_i, y_i), which equals:

$$\frac{|ax_i + by_i + c|}{\sqrt{a^2 + b^2}} = |ax_i + by_i + c|$$

Problem Formulation

From the above, we can formulate a concise definition of the optimization problem that we want to solve for any set of points P. Assuming the standard least-squares criterion that aims at minimization of the squared distances from each points to the resulting line, we wish to minimize the following function:

$$\sum_{(x,y)\in P} (ax + by + c)^2$$

over all lines (a, b, c) that meet the criterion $\sqrt{a^2 + b^2} = 1$.

Moment-based Solution

Interestingly, it turns out that we have already discussed almost identical problem, although stated rather covertly. Minimization of the squared distances from a point set to a line is, under physical terminology, nothing else that finding the axis of minimal inertia of the point set; as its moment of inertia equals the sum of squared distanced from each point to the axis of rotation.

As we have already indicated in the **Contour Analysis** chapter, such axis may be computed as the eigenvalue of the inertia tensor of the rigid body:

$$I = \begin{bmatrix} c'_{2,0} & c'_{1,1} \\ c'_{1,1} & c'_{0,2} \end{bmatrix}$$

All that we need to do to adapt this solution to our needs is to define the moments of a point set P, which are directly equivalent to the moments which we have defined for regions:

$$m_{p,q} = \sum_{(x,y)\in P} x^p y^q$$
$$c_{p,q} = \sum_{(x,y)\in P} (x-\overline{x})^p (y-\overline{y})^q$$

Equivalent result may be also obtained directly using multivariate analysis - the derivation may be found in [16, p. 588-591].

Figure 6.3 demonstrates example results of least-squares line fitting.

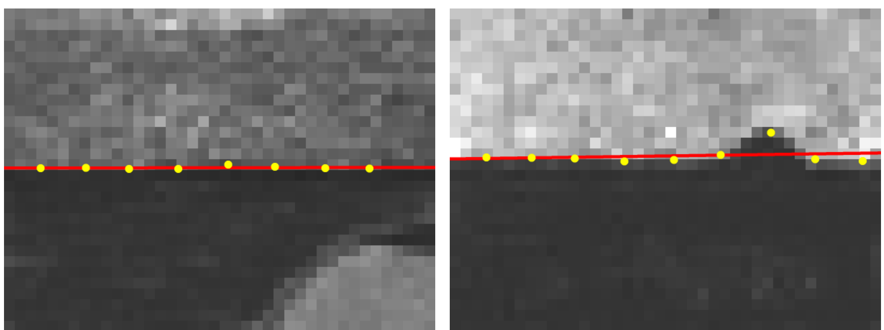


Figure 6.3: Example results of least-squares line fitting.

Outlier Suppression

It is often the case that, due to the imperfections of edge extraction or the object being analyzed, the input data contains points that do not represent accurately the shape that we wish to extract; as demonstrated in the image on the right of **Figure 6.3**. Such points, called outliers, distort the outcome of the fitting and should be skipped or suppressed in the computation.

Random Sample Consensus

Random Sample Consensus, proposed[21] by Fischler and Bolles, is a popular, probabilistic method of robust estimation that exhibits good performance against outliers, even when they represent a significant fraction of the data.

The method iteratively selects a random minimal sample of points that allows an estimation of the results, in our case that would be a pair of points, to which a line L is fitted. After each such estimation the algorithm counts the data points that *support* the estimate, i.e. lie within a predefined distance d from L.

The algorithm terminates after a predefined number of iterations, fitting a line to all points that supported the most supported estimate.

The major drawback of such approach in the context of industrial applications is its probabilistic nature - for the robustness of the systems being designed it is crucial that the methods being deployed should succeed and fail deterministically and consistently on similar data instances.

Iterative Suppression

Another, more predictable in its outcomes, approach would be to improve the results of the fitting iteratively, progressively suppressing the influence of the points that are distant from the estimates being obtained.

After each least-squares fit yielding a line L, we may calculate the distances between each data point and L and inspect the distribution of distances looking for possible outliers. If we compute a nonnegative weight w_i for each point reflecting our belief that the point i is not an outlier, we could limit the influence of the points with low weight using weighted moments defined as follows:

$$
\begin{aligned}
m_{p,q} &= \sum_{(x,y,w)\in P} wx^p y^q \\
m'_{p,q} &= \frac{1}{t} m_{p,q} \\
c_{p,q} &= \sum_{(x,y,w)\in P} w(x-\overline{x})^p (y-\overline{y})^q \\
c'_{p,q} &= \frac{1}{t} c_{p,q}
\end{aligned}
$$

where $t = \sum_{(x,y,w)\in P} w$ is a sum of all weights. Such values would be used instead of the classic moments in the successive iteration of the algorithm.

There are a number of possibilities regarding the specific methods of calculating the weights w_i given the distances d_i between each point and the last estimate of the line. A classic method proposed by Huber computes the weights as:

$$
w_i = \begin{cases} 1 & d_i \le t \\ \frac{t}{d_i} & d_i > t \end{cases}
$$

where the threshold t may be either calculated from the distribution of distances d_i (e.g. as its median) or be left as a parameter of the algorithm. Example results of iterative outlier suppression are demonstrated in **Figure 6.4**.

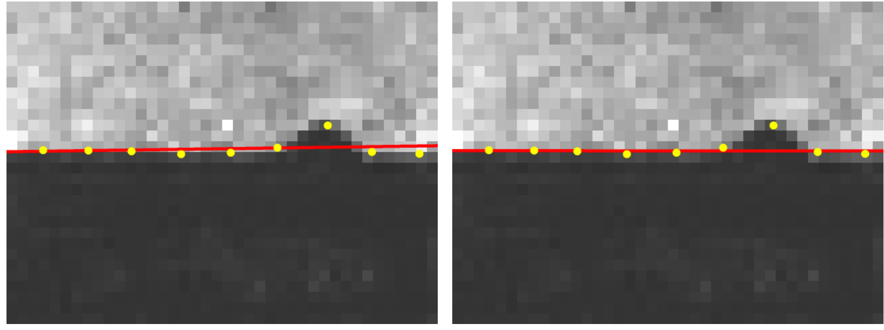


Figure 6.4: Example results of line fitting without and with outlier suppression.

Segments

Line fitting may be directly applied to fit segments to data points, as we may start with fitting a line to the points and then compute the resulting segment as a section of the line defined by the orthogonal projection of the input points onto the line.

> **Adaptive Vision Studio 4** filter `FitLineToPoints` implements the line fitting along with optional iterative outlier suppression, while the `FitSegmentToPoints` performs analogous segment fitting.

6.3 Circles

We now move our attention to the problem of circle fitting. A canonical equation used to represent circles is the following:

$$\{(x, y) : (x - a)^2 + (y - b)^2 = r^2\}$$

where (a, b) is the center of the circle and r denotes its radius. Under this representation, distance between the circle and a point (x_i, y_i) is given by:

$$|\sqrt{(x_i - a)^2 + (y_i - b)^2} - r|$$

The least-squares circle fit to a point set P may be therefore formulated as the minimization of the error:

$$\sum_{(x,y)\in P} (\sqrt{(x - a)^2 + (y - b)^2} - r)^2$$

over possible circles (a, b, r).

Unfortunately, this problem is nonlinear and cannot be solved by a finite algorithm. The existing methods for circle fitting may be divided into two groups: iterative algorithms that optimize the least-squares function defined above (these methods are usually referred to as *geometric*) and methods that optimize other, easier to solve, cost functions (these methods are usually called *algebraic*).

The geometric fit is theoretically enticing, but causes significant problems in practical applications. Every algorithm trying to find the least-squares fit is inherently prone to divergence or convergence to suboptimal local minimum. Moreover, the geometric fit algorithms tend to be computationally expensive, at least when compared with the popular methods of algebraic fitting.

At the same time the best available methods of algebraic fitting yield accurate, close-to-optimal results and are free from this limitations - both experimental and theoretical arguments on this issue can be found in an exhaustive work[22] of Chernov.

Algebraic Fit

Perhaps the simplest method of algebraic circle fitting was given by Kasa, who proposed to minimize the following error function:

$$\sum_{(x,y)\in P} ((x-a)^2 + (y-b)^2 - r^2)^2$$

After a simple substitution $A = -2a$, $B = -2b$, $C = a^2 + b^2 - r^2$ and introducing artificial, third dimension of the point set $z = x^2 + y^2$ we get:

$$\sum_{(x,y)\in P} (z + Ax + By + D)^2$$

The minimum of this function can be found by computing the partial derivatives with respect to A, B, C - we require that each such derivative is equal to zero at the minimum, which leads to the following system of linear equations:

$$\begin{aligned} m'_{2,0,0}A + m'_{1,1,0}B + m'_{1,0,0}C &= -m'_{1,0,1} \\ m'_{1,1,0}A + m'_{0,2,0}B + m'_{0,1,0}C &= -m'_{0,1,1} \\ m'_{1,0,0}A + m'_{0,1,0}B + C &= -m'_{0,0,1} \end{aligned}$$

where $m'_{p,q,s}$ denotes three-dimensional (together with the artificial dimension z), normalized moments. Such system can be solved efficiently by methods of linear algebra.

Kasa's method has a number of advantages: it is very fast, it is easy to implement, and its moment-based schema is easy to equip with the iterative outlier suppression which we have described in the last section. Its major weakness is poor performance on data representing small arcs (10 degrees or less), for which it often fails to yield a feasible estimation.

Fortunately, more advanced methods of algebraic fitting by Pratt and Taubin achieve far better performance retaining the advantages of the algebraic fit. Details of these methods as well as a benchmark of their performance are given in the already mentioned work[22], which indicates the Taubin method as feasible for the majority of practical applications.

Figure 6.5 demonstrates the results of Kasa and Taubin fit for the data set representing small circular arc.

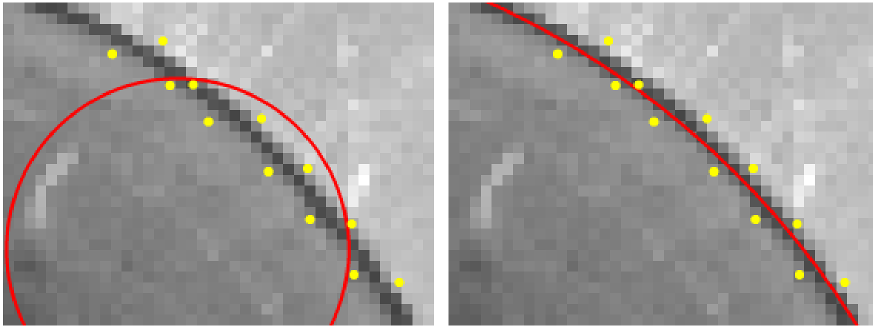


Figure 6.5: The algebraic fit of Kasa (on the left) and Taubin (on the right).

Circular Arcs

Circle fitting may be easily modified to return circular arcs rather than full circles. Again, all that we need to do is to compute the orthogonal projection of the input points onto the resulting circle and select the smallest section of the circle that contains all projection points.

Adaptive Vision Studio 4 filter `FitCircleToPoints` implements a selection of algebraic line fitting methods along with optional iterative outlier suppression, while the `FitArcToPoints` performs analogous fitting of circular arcs.

6.4 Fitting Approximate Primitives to Images

Utilizing the shape fitting techniques in the context of image analysis requires extraction of the points to which the primitives will be fitted. Typically such extraction is done by means of **1D Edge Detection** or **2D Edge Detection**, which allow to fit the shapes to step edges or ridges present in the image.

The methods of **1D Edge Detection** allow to build particularly useful technique for fitting approximately positioned primitives to their actual occurrences. The idea is to take a rough estimation of the primitive location and construct a set of scan lines going across the approximate primitive. By performing one-dimensional edge or ridge detection along each scan line, we obtain a set of points that, hopefully, represent the actual position of the primitive being located.

In **Figure 6.6** a segment is fitted to the edge of an object, while **Figure 6.7** demonstrates an example of fitting a circle to a circular ridge.

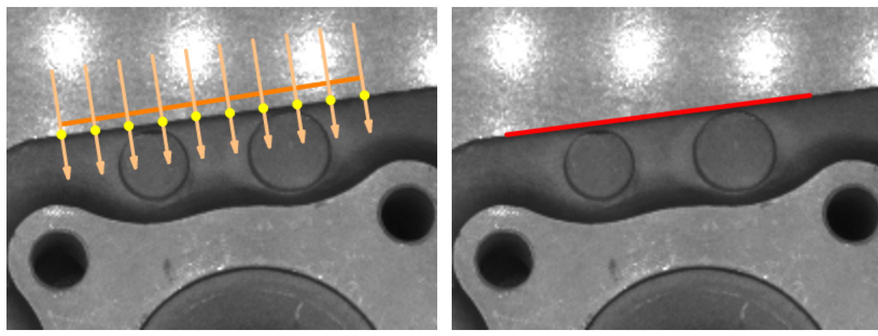


Figure 6.6: Shape fitting applied to fit a segment to an edge.

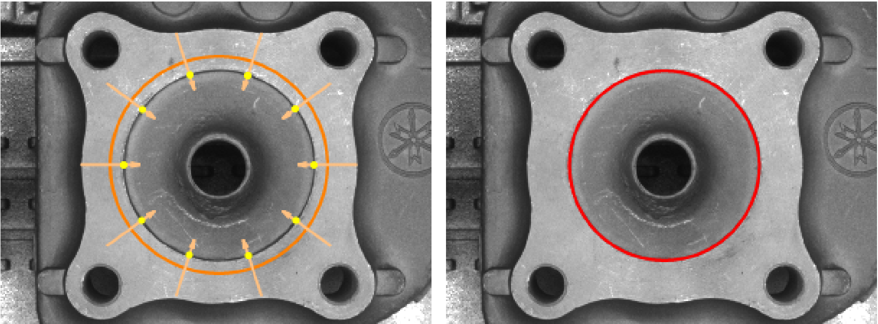


Figure 6.7: Shape fitting applied to fit a circle to a ridge.

Adaptive Vision Studio 4 provides filters for each type of primitive, one filter of a pair fitting the primitive to image edges, second - to image ridges, and third - to image stripes:

- `FitArcToEdges`, `FitArcToRidge`, `FitArcToStripe`
- `FitCircleToEdges`, `FitCircleToRidges`, `FitCircleToStripe`
- `FitPathToEdges`, `FitPathToRidges`, `FitPathToStripe`
- `FitSegmentToEdges`, `FitSegmentToRidges`, `FitSegmentToStripe`

6.5 Examples

Measurements

Figure 6.8 demonstrates an application of the shape fitting in which it is used to measure the angle between two edges of a saw tooth.

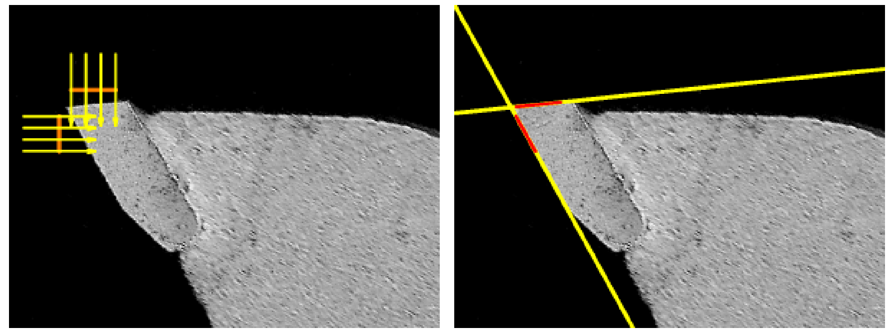


Figure 6.8: An example application of shape fitting.

Chapter

7

Template Matching

> If Edison had a needle to find in a haystack, he would proceed at once with the diligence of the bee to examine straw after straw until he found the object of his search.
>
> A little theory and calculation would have saved him ninety per cent of his labor.
>
> — Nikola Tesla

7.1 Introduction

We have already discussed a number of methods that may be utilized for object localization. For instance - we can use simple **Image Thresholding** (possibly together with **Blob Analysis**) to identify objects of contrasting brightness. **1D Edge Detection** may be applied to locate object boundaries whenever we already know its approximate position. **Contour Analysis** allows to identify objects by analysing the paths which we have previously extracted using **2D Edge Detection** etc.

While these methods allow to construct **tailored solutions** for particular applications, identification problems may turn out to be too complex to allow for a convenient application of any of the above. Moreover, specialized solutions are inherently coupled with the particular object to be found, which makes it hard to update the inspection system when the problem specification changes (e.g. because one of the components used in the production is replaced by another).

In this chapter we will discuss the most general-purpose technique of object localization - **template matching**, which allows to identify parts of an image that match, under some criterion of similarity, an arbitrarily chosen image template. Such methods not only allow to solve identification problems that otherwise could not be tackled, but also provide a convenient (yet often more computationally expensive) alternative to the methods that we have discussed before.

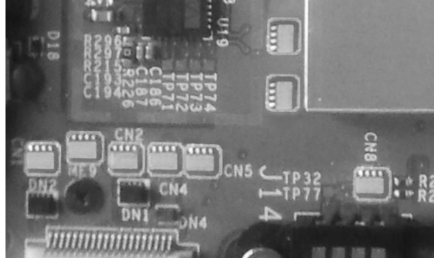


Figure 7.1: An example input for a template matching problem - occurrences of the template depicted on the left are to be found in the image of the right.

The most important part of any template matching method is the specific measure of image similarity that will be used to evaluate possible matches. We need such measure not only to design the algorithm, but also to define the task in the first place - if we are supposed to find the template occurrences, we need to specify what does it mean that a template *occurs* at some position in an image.

A template aligned over an image at some position corresponds to a section of the image that it overlaps. If we assume that the possible template alignments are pixel-precise, the problem of evaluating the quality of the match is essentially a problem of calculating the similarity of two images of equal dimensions.

In this chapter we will discuss two groups of template matching techniques, differing in image similarity measure they are built upon:

- **Brightness-based matching** evaluates the image similarity using brightness properties of the images.
- **Edge-based matching** compares the gradient-based features of both images.

7.2 Brightness-Based Matching

Brightness-based matching evaluates possible template alignments using measures of image similarity based on the relation between the brightness of the corresponding pixels of the images being compared.

Image Correlation and Image Difference

While some of the measures of image similarity have natural formulation which yields high values for well-correlated images and low values for images that differ significantly, other are easier to define conversely, as measures of difference between images. To distinguish between these two groups we will use the term **image correlation** when referring to direct measures of image similarity and **image difference** when considering measures of dissimilarity. Naturally, both are equally suitable for our needs.

We will discuss three important measures of image similarity in the context of a simple comparison benchmark demonstrated in **Table 7.1**.

			ME	MSE	NCC
(a)			54.196	3933.365	0.090
(b)			8.455	623.833	0.702
(c)			10.894	271.555	0.866
(d)			18.528	550.718	1.000

Table 7.1: Image similarity measures evaluated on example image pairs.

Mean Error (ME)

Perhaps the simplest measure of image difference is the average of its pixel-wise differences, i.e.:

$$\text{ME}(\text{Image}_1, \text{Image}_2) = \frac{1}{n} \sum_{x,y} |\text{Image}_1[x,y] - \text{Image}_2[x,y]|$$

where n denotes the number of pixels within the image dimensions.

It is important to note that the response of this measure to a single difference depends linearly on its magnitude. This is not an optimal behavior for template matching purposes - we should use a measure that will amplify strong differences, otherwise a big number of small differences (which are inevitable due to the noise and lightning imperfections) will have impact on the result comparable with a small number of extreme differences (which should weight significantly on the score of a possible match).

This issue may be noticed in **Table 7.1** examples (b) and (c). The first one yields lower mean error, even though it misses important part of the template while the second one matches the structure of the template perfectly, but contains fine differences over its entire area.

Mean Squared Error (MSE)

This problem is easy to address directly, by powering each term of the average:

$$\mathrm{MSE}(\mathrm{Image}_1, \mathrm{Image}_2) = \frac{1}{n} \sum_{x,y} (\mathrm{Image}_1[x,y] - \mathrm{Image}_2[x,y])^2$$

Indeed, we may notice that **MSE** of the benchmark pair (c) is indeed less than half of **MSE** of (b).

This measure is already feasible for applications in which the lightning may be relied on to be constant and uniform. Unfortunately, any change in the image brightness will heavily increase both **ME** and **MSE**. This effect may be noticed in the results for benchmark dataset (d), the second image of which is the result of linear transformation $(0.5x - 100)$ of the first image.

Normalized Cross-Correlation (NCC)

To overcome this issue we will move our attention towards an important measure of image correlation - normalized cross-correlation.

Normalized cross-correlation is based on an interpretation of an image as a simple one-dimensional vector of its n pixel values. Such interpretation is accurate as long as we compute the image similarity pixel-by-pixel, disregarding the spatial positions of the pixels.

For meaningful comparison of two such vectors, we need to subtract mean brightness of each image from its pixel values, thus eliminating the additive factor that would distort the proportions between individual dimensions of each vector. After such subtraction we will be left with two vectors, each having n elements of the form $\mathrm{Image}[i,j] - m$.

Having eliminated the additive factor from each image, we may now deal with the multiplicative factor by normalizing each vector, i.e. by division of each element by the vector length computed as $\sqrt{\sum_{i,j}(\text{Image}[i,j]-m)^2}$, which happens to be the exact definition of the standard deviation σ of the original image pixel brightness.

Once we have reduced each image to the form of normalized, n-dimensional vector, we may evaluate their similarity by computing the angle θ between them. Such angle may be easily obtained, as the dot product of two normalized vectors $\mathbf{a}$ and $\mathbf{b}$ equals the cosine of the angle between them:

$$\cos(\theta) = \sum_{i=1}^{n} a_i b_i$$

We may actually leave the result in the form of $\cos(\theta)$ as the cosine will conveniently scale the result to the $(-1.0, 1.0)$ range. Putting the pieces together, we obtain the formula for the normalized cross-correlation of two images:

$$\text{NCC}(\text{Image}_1, \text{Image}_2) = \frac{1}{n\sigma_1\sigma_2} \sum_{x,y} (\text{Image}_1[x,y] - m_1)(\text{Image}_2[x,y] - m_2)$$

The obtained method is invariant to linear changes in image brightness and proves suitable on the benchmark data set. From now on we will assume this is the method that our template matching algorithm will be based on.

Search Procedure

Having selected a feasible brightness-based measure of image similarity, we may proceed to discussion of the search process itself.

Template Correlation Image

An exhaustive search for the template occurrences is straightforward to define - we may consider each possible alignment of the template over the image and compute the normalized cross-correlation factor between the template and the part of the image that it overlaps. Results of such search may be conveniently represented on an image, each pixel of which represents the similarity

factor of the template aligned over the pixel. Example *template correlation image* for the input data from **Figure 7.1** is demonstrated in **Figure 7.2**.

Figure 7.2: Example template correlation image. Each pixel takes a value between −1.0 (black) and 1.0 (white).

Such image allows to identify the matching locations in the classic way resembling the criteria that we have used to find edge points in the **1D Edge Detection** chapter: by extracting locally maximal points (non-maximum suppression) of magnitude greater than a predefined threshold. Example results are demonstrated in **Figure 7.3**.

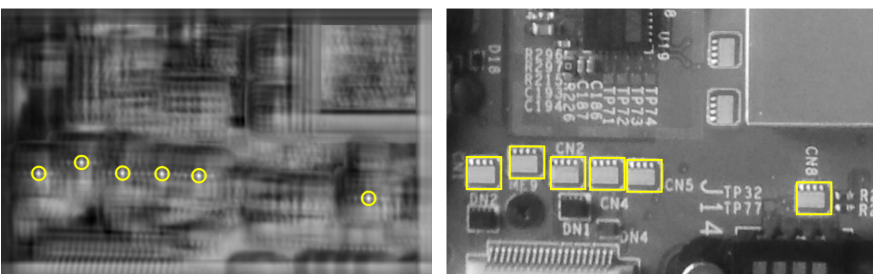


Figure 7.3: Local maxima of the example template correlation image stronger than 0.75 on the left, corresponding matches in the original image on the right.

Pyramid Matching

A major drawback of such simplistic search for the template occurrences lies in the computational cost of calculating the normalized cross-correlation at

each location. We can trade the precision of the search for speed if we reduce the resolution of the images that we are to work with, but then we would have to face the problem of low precision, which is often equally undesirable as the problem of high processing time.

Let us define the notion of *image pyramid* as a series of images, each image being a reduced-resolution counterpart of the previous image. Usually the resolution is reduced by the factor of two in each dimension, as demonstrated in **Figure 7.2**. The elements of image pyramid are conventionally referred to as levels, the original image being the first element at level 0.

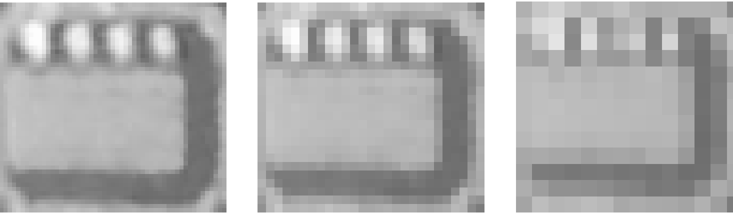


Table 7.2: Example image pyramid.

Pyramid matching is a technique proposed[23] by Tanimoto, that utilizes image pyramids to combine advantages of high-precision search for template occurrences in the original images with the benefits of high-speed search in the downsampled images.

Pyramid search commences by computing pyramids of both the input images. The height of the pyramid is a parameter of the algorithm. It should be set to maximum value for which the template is still recognizable on the highest level of its pyramid.

Once the image pyramids are computed, the exhaustive search for template occurrences is run using significantly downsampled images on the highest pyramid level. Similar search is then repeated at successive pyramid levels down to the original images, however on each level only the template positions that scored high on the previous level are considered.

In this way the image areas that are unlikely to contain template occurrences are pruned from the computation on the coarse level, while promising positions are pursued through every level and in the end are identified with high precision on the original images.

Multi-angle Matching

So far we have been assuming that the template to be identified has a fixed orientation in relation to the image axes, which is rarely the case in practice. The search process that we have described has to be therefore extended to allow for rotations of the template.

To achieve that, we will add two additional parameters to the specification of the problem instance: a range of allowed orientations and the angular precision of the search. For instance, angle range $(-15°, 15°)$ and angle precision $1.0°$ would mean that 31 rotations of the input template shall be taken into account in the search routine. For each such rotations, separate pyramid of the template image will be computed.

In the search procedure itself we need to identify possible match candidates using tuples of $(position, rotation)$, rather than sole positions. As such extension of the pyramid search increases the computational load roughly by the factor of the number of rotations to be considered, it is advisable to limit the range of angles being considered whenever possible (e.g. when technical conditions prevent significant deviations of the template occurrences orientation). Example results obtained using this technique are presented in **Figure 7.4**

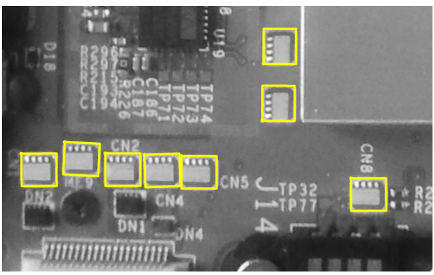


Figure 7.4: Results on the multi-angle matching of the example data.

Similar extension may be used to match the template on multiple scales, yet such need is less common than the multi-angle matching.

Offline Phase

From the above description of pyramid matching it is clear that certain amount of preprocessing is required both for the template and the search image. It is important to note, that the computational load of preprocessing for these two images is **not** equally important.

It is extremely rare to perform template matching with one-off template - usually the template is fixed and used for inspection of a whole series of images, e.g. being captured and analyzed online. The preprocessing of the template image almost always may be performed offline and as such does not need to be the focus of possible optimizations. The results of such preprocessing along with the template image are often saved in a form of atomic datatype for easy reuse.

The preprocessing of the search image on the other hand in most cases has to be performed online and therefore its efficiency may be crucial for the feasibility of the whole template matching-based solution. It is worth noting that the extension of the pyramid matching to multiple orientations requires computing of separate pyramids of the **template image** for every possible rotation, while the preprocessing of the search image remains limited to the calculation of a single pyramid.

The measures of image similarity are implemented in **Adaptive Vision Studio 4** filters:

- `ImageCorrelation`, `ImageCorrelationImage`
- `ImageDifference`, `ImageDifferenceImage`

First filter of each pair computes a single similarity value for two images of equal dimensions, while the second filter performs a template similarity evaluation of each possible template alignment and represents the results in form of an image, as previously described.

Brightness-based pyramid matching is implemented in three **Adaptive Vision Studio 4** filters: `CreateGrayModel`, `LocateSingleObject_NCC` and `LocateMultipleObjects_NCC` - first filter performs the offline, template preprocessing phase and stores the results in atomic, reusable datatype called *model*, which can be later used to perform the actual matching using the second and third filter.

7.3 Edge-Based Matching

While the properties of normalized cross-correlation and efficiency boost offered by the pyramid matching make decent implementations of brightness-based template matching suitable for a range of typical applications, the algorithm retains a few weaknesses such as sensitivity to non-linear illumination changes and low capability of matching partially visible (occluded) template occurrences.

As we have already noted, image edges are usually well preserved under disturbances of the illumination. Moreover, edges of an object define precisely its shape and therefore are a suitable discriminant for the identification of possible matches. The idea of edge-based template matching has been subject to extensive research and numerous measures of edge-based similarity between two images have been proposed.

Symmetric Methods

Perhaps the most fundamental classification of the edge-based template matching methods would be based on the relation between the template image and search image processing being performed by the algorithm. The earliest edge-based template matching methods were built upon edge extraction performed in the same way on both images.

One of the earlies methods was proposed[24] by Borgefors, who suggested to extract the edge pixels in both images and evaluate the matches using the **mean squared distance** between each edge pixel of the template image and the nearest edge pixel in the search image.

Rucklidge proposed[25] a similar measure based on the **Hausdorff distance** between the edges extracted in both images, in which the difference between two images equals the maximum of two distances:

- Longest distance between a template image edge pixel and any of the search image edge pixels.
- Longest distance between a search image edge pixel and any of the template image edge pixels.

The major weakness of these and similar methods lies in the very idea of symmetric edge extraction. While the accurate edge extraction in the template image should not pose a problem, the assumption that the method which worked for the template image will consistently work for each search image, possibly under variable lightning conditions, is by far overoptimistic.

Moreover, both methods are inherently sensitive to occlusions and as such have few, if any, advantages over the normalized cross-correlation grayscale-based template matching.

Asymmetric Methods

To overcome the limitations of the symmetric edge-matching we need to move towards asymmetric methods. Steger proposed[26] to perform full edge detection in the template image, precisely identifying its contours; while each search image would be subject only to simple gradient computation on its entire area. Each possible match is then evaluated based on the correspondence of the gradient direction between the template and the search image, but only at the positions of template edge pixels.

This idea is demonstrated in **Figure 7.5**, where we use colors from the HSV circle to represent gradient directions.

Such approach immediately eliminates the problem of accurate edge extraction in the search image, as no such extraction is performed. Moreover, it utilizes matching of the edge directions, not only their presence, which was earlier demonstrated[27] by Olson et al. as advantageous in eliminating false-positive results in the case of Hausdorff distance metric.

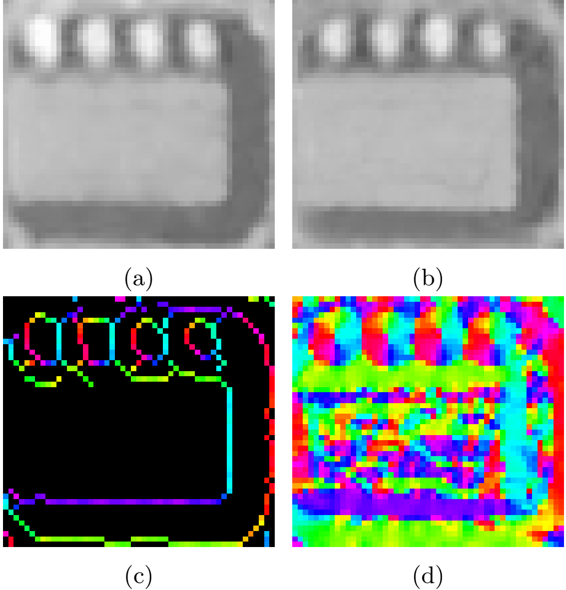


Figure 7.5: Asymmetric edge processing - template image (a) is subject to full edge detection after which gradient direction at its edge pixels is computed (c), while for the corresponding section of the search image (b) computation of the gradient direction is performed indiscriminately on its entire area (d).

The specific method of calculating the score of a match proposed by Steger is the simple sum of cosines of the angles between the corresponding gradient vectors, resembling the idea behind the normalized cross-correlation measure.

Matching based on gradient direction does not preclude the application of the pyramid schema. It is interesting to note that the asymmetric method actually outperforms the traditional grayscale-based template matching in terms of speed, as the evaluation of each possible match is limited to the edge pixels of the template image; while the additional burden of computing the gradient of search image pyramid is not significant (compared with the dominating cost of the match evaluation).

The method also works well against occlusions, as each missing edge pixel in the search image contributes at most the penalty of -1 to the final score, which equals in magnitude the input of a perfectly matched gradient direction, yielding cosine value of 1. A detailed evaluation of the performance of this method is given in [28].

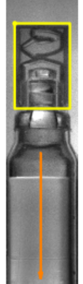
> Edge-based template matching is implemented in three **Adaptive Vision Studio 4** filters: `CreateEdgeModel`, `LocateSingleObject_Edges` and `LocateMultipleObjects_Edges`

7.4 Examples

Positioning of Scope Primitives

Template matching techniques are commonly applied to position scan lines, regions of interest, prototype shapes, etc. over an appropriate element of the object being inspected. To do so, a characteristic object location of a fixed position in relation to the area being measured is selected, and the scope primitives are defined relatively to the characteristic location.

Then, during the actual inspection, characteristic locations of the object instances are identified using template matching techniques, and the scope primitives are positioned accordingly, as demonstrated in **Figure 7.6**.



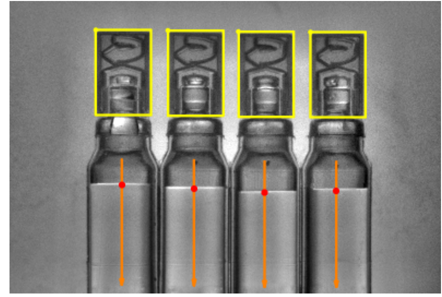


Figure 7.6: Template matching used for alignment of the scan lines for **1D Edge Detection**.

Bibliography

[1] Mehmet Sezgin and Bülent Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *J. Electronic Imaging*, 13(1):146–168, 2004.

[2] Judith Prewitt and Mortimer Mendelsohn. The analysis of cell images. *Annals of the New York Academy of Sciences*, 128:1035–1053, 1966.

[3] J. N. Kapur, P. K. Sahoo, and A. K. C. Wong. A new method for gray-level picture thresholding using the entropy of the histogram. *Computer Vision, Graphics, and Image Processing*, 29:273–285, 1985.

[4] T. W. Ridler and S. Calvard. Picture thresholding using an iterative selection method. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-8(8):630–632, 1978.

[5] N. Otsu. A threshold selection method from gray level histograms. *IEEE Trans. Systems, Man and Cybernetics*, 9:62–66, 1979.

[6] Pierre Soille. *Morphological Image Analysis: Principles and Applications.* Springer-Verlag, 2nd edition, 2003.

[7] Azriel Rosenfeld. Connectivity in digital pictures. *Journal of the ACM*, 17(1):146–160, 1970.

[8] Carsten Steger, Markus Urlich, and Christian Wiedemann. *Machine Vision Algorithms and Applications*. WILEY-VCH, 2008.

[9] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[10] Ramesh Jain, Rangachar Kasturi, and Brian Schunck. *Machine Vision*. MIT Press and McGraw-Hill, 1st edition, 1995.

[11] J. Brian Subirana-Vilanova and Kah Kay Sung. Ridge-detection for the perceptual organization without edges. In *Fourth International Conference on Computer Vision*, pages 57–64, 1993.

[12] Mark Nixon and Alberto Aguado. *Feature Extraction & Image Processing*. Academic Press, 2nd edition, 2008.

[13] Otto Schmitt. A thermionic trigger. *Journal of Scientific Instruments*, 1938.

[14] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1:244–256, 1972.

[15] Ken Turkowski. Computing 2d polygon moments using green's theorem. *Apple Technical Report*, 1997.

[16] R. M. Haralick and L. G. Shapiro. *Computer and Robot Vision*, volume 1. Addison-Wesley, 1992.

[17] Godfried Toussaint. Solving geometric problems with the rotating calipers, 1983.

[18] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[19] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, pages 216–219, 1979.

[20] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In Hermann Maurer, editor, *Proceedings of New Results and New Trends in Computer Science*, volume 555, pages 359–370, Berlin, Germany, 1991. Springer.

[21] Martin Fischler and Robert Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[22] Nikolai Chernov. *Circular and Linear Regression: Fitting Circles and Lines by Least Squares*. 1st edition, 2010.

[23] Steven Tanimoto. Template matching in pyramids. *Computer Graphics and Image Processing*, 16(4):356–369, 1981.

[24] G. Borgefors. Hierarchical chamfer matching: a parametric edge matching algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10:849–865, 1988.

[25] W. J. Rucklidge. Locating objects using the hausdorff distance. In *ICCV*, pages 457–464, 1995.

[26] Carsten Steger. Occlusion, clutter, and illumination invariant object recognition. In *Photogrammetric Computer Vision*, 2002.

[27] C. F. Olson and D. P. Huttenlocher. Automatic target recognition by matching oriented edge pixels. *IEEE Trans. Image Processing*, 6(1):103–113, January 1997.

[28] Markus Ulrich and Carsten Steger. Performance comparison of 2D object recognition techniques. In *Photogrammetric Computer Vision*, 2002.