Algorithms in Digital Geometry Based on Cellular Topology

V. Kovalevsky

University of Applied Sciences Berlin

www.kovalevsky.de; kovalev@tfh-berlin.de

Abstract. The paper presents some algorithms in digital geometry based on the topology of cell complexes. The paper contains an axiomatic justification of the necessity of using cell complexes in digital geometry. Algorithms for solving the following problems are presented: tracing of curves and surfaces, recognition of digital straight line segments (DSS), segmentation of digital curves into longest DSS, recognition of digital plane segments, computing the curvature of digital curves, filling of interiors of *n*-dimensional regions (*n*=2,3,4), labeling of components (*n*=2,3), computing of skeletons (*n*=2, 3).

1 Introduction

First of all let us discuss the question, why should one use cell complexes in digital geometry. The author supposes that there are three categories of researchers in this field. Those of the first category would prefer a mathematical proof of the assertion that cell complexes belong to the class of locally finite topological spaces that are in agreement with the axioms and with most definitions of the classical topology.

Researchers of the second category would perhaps ask, whether classical axioms are really important for applications. They may believe that it is possible to find another set of axioms and deduce the properties of the topological space best suitable for applications in image analysis and in computer graphics from these new axioms.

Researchers of the third category pay no attention at all to axiomatic theories. They are only interested in methods enabling one to develop efficient solutions of geometric and topological problems in image analysis and in computer graphics.

The author hopes to satisfy with the present paper the desires of researchers of all three categories.

Let us start with the second category. First of all it is necessary to mention that we don't need considering spaces in which each neighborhood of a point contains an infinite number of points, since such a space cannot be explicitly represented in a computer. A space in which each element has a neighborhood containing a finite number of elements is called a *locally finite space* (LFS).

The author believes that everybody will agree that the features of a space, most important for applications are those of connectivity and of boundary. Therefore we suggest the following set of axioms concerned with these notions. We denote the elements of the space not as "points" since, as we shall see soon, there are in an LFS elements with different neighborhoods. They have different topological properties and thus must have different notations.

Axiom 1: For each space element *e* there are certain subsets containing *e*, which are neighborhoods of *e*. Since the space is locally finite there exists the smallest neighborhood of *e*.

We shall denote the smallest neighborhood of e by SON(e). The exact definition of the neighborhood will be derived from the whole set of the Axioms below.

Axiom 2: There are space elements whose SON consists of more than one element. If a and b are space elements and $b \in \mathrm{SON}(a)$ then the set $\{a, b\}$ is connected. Also a set $\{a\}$ consisting of a single space element is connected.

We shall say that a and b are *directly connected* or *incident* to each other. This is a binary relation *Inc*. Since $\{a, b\}$ and $\{b, a\}$ denote one and the same set, the incidence relation *Inc* is symmetric. It is reflexive since according to Axiom 2 the set $\{a\}$ is connected.

Axiom 3: The connectivity relation is the transitive hull of the incidence relation. It is symmetric, reflexive and transitive. Therefore it is an equivalence relation.

Let us now formulate the axioms related to the notion of a boundary. The classical definition of a boundary (exactly speaking, of the topological boundary or of the frontier) is as follows:

Definition BD: The topological boundary of a subset T of the space S is the set of all space elements whose *each* neighborhood intersects both T and its complement $S-T$.

In the case of a locally finite space it is obviously possible to replace "each neighborhood" by "smallest neighborhood". We shall denote the topological boundary i.e. the frontier of the subset T of the space S by $\mathrm{Fr}(T,S)$. Now we introduce the notion of a *thin* boundary:

Definition TB: The boundary $\mathrm{Fr}(T,S)$ of a subset T of an n-dimensional space S is called *thin* if it contains no n-dimensional cube of 2^n mutually incident space elements.

Axiom 4: The topological boundary of any subset T is thin and is the same as the topological boundary of the complement $S-T$.

Let us remain the reader the classical axioms of the topology. The topology of a space S is defined if a collection of subsets of S is declared to be the collection of the *open subsets*. These subsets must satisfy the following Axioms:

Axiom C1: The whole set S and the empty subset $\varnothing$ are open.

Axiom C2: The union of any number of open subsets is open.

Axiom C3: The intersection of a finite number of open subsets is open.

Axiom C4: The space has the separation property.

There are (at least) three versions of the separation property and therefore three versions of Axiom C4 (Fig. 1.1) :

Axiom T_0: For any two distinct points x and y there is an open subset containing exactly one of the points.

Axiom T_1: For any two distinct points x and y there is an open subset containing x but not y and another open subset, containing y but not x.

Axiom T_2: For any two distinct points x and y there are two non-intersecting open subsets containing exactly one of the points.

A space with the separation property T_2 is called *Hausdorff space*. The well-known Euclidean space is a Hausdorff space.

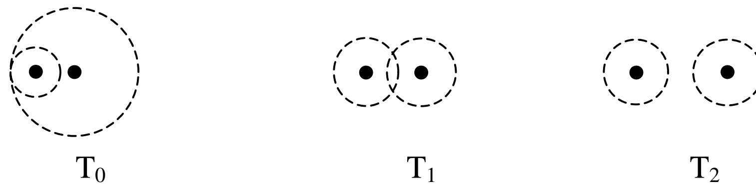


Fig. 1.1. A symbolic illustration to the separation axioms

It is easily seen that if a single point is not an open subset, then only the Axiom T_0 may be applied to a locally finite space (LFS). Really, Axioms T_1 and T_2 demand that the open subsets under consideration contain infinitely many points, no matter how small they are. Such subsets do not exits in a LFS and cannot be explicitly represented in a computer. Therefore only Axiom T_0 is relevant for an LFS.

The author has proved [11] that the above set of Axioms C1, C2, C3 and T_0 is equivalent to the set of our suggested set of Axioms 1 to 4. This means that Axioms C1 to C3 and T_0 may be deduced from Axioms 1 to 4 as theorems. This proves that the classical notion of open subsets is important for providing a space with the features of connectivity and boundary that satisfy the "obviously true" demands formulated as our Axioms 1 to 4.

Unfortunately, it is impossible to repeat here the proof because of lack of space. We shall only show that the classical Axiom T_0 follows from the demands that the boundaries be thin and that the boundary of a subset T be the same as the boundary of its complement $S–T$.

Theorem NT. If the neighborhood of a space element e in the Definition BD contains besides e itself all elements that stay in a *symmetric* binary relation with e then there exist subsets whose boundary is not thin.

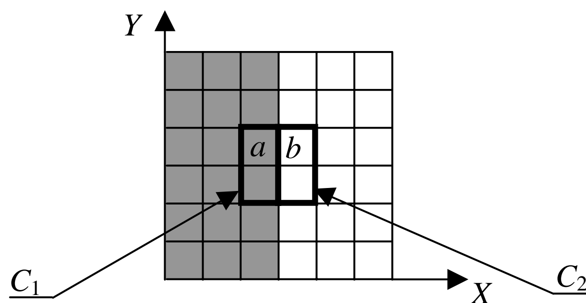


Fig. 1.2. An illustration to the proof of Theorem NT for n=2

Proof: We presume that two elements whose one coordinate differs by 1 while all other coordinates are equal satisfy the symmetric relation mentioned in Theorem NT. Let T be a subset of the space S that contains an $(n-1)$-dimensional cube $C_1=\{x_1=m\}\times\{x_2, x_2+1\}\times...\times\{x_n, x_n+1\}$ while all elements of the "adjacent" cube

$C_2=\{x_1=m+1\}\times\{x_2, x_2+1\}\times...\times\{x_n, x_n+1\}$ belong to the complement $S-T$ (Fig. 1.2). The neighborhood of any element $a\in C_1$ contains the element $b\in C_2$ whose coordinate x_1 is equal to $m+1$ and all other coordinates are equal to the corresponding coordinates of a. Thus $b\in S-T$ and a belongs to the boundary of T. Since the relation defining the neighborhoods is symmetric, a belongs to the neighborhood of b, that intersects T at a and $S-T$ at b. Therefore b belongs to the boundary of T. This is true for all elements of the cube C_2. Thus both cubes C_1 and C_2 are in the boundary of T. Their union is an n-dimensional cube and hence the boundary is not thin.■

There are two possibilities to achieve that the boundary be thin for any subset:

1. To change the Definition BD of the boundary so that only elements of T may belong to the boundary of T.
2. To define the neighborhood by means of an *antisymmetric relation*, which means that if b is in the smallest neighborhood of a then a is not in the smallest neighborhood of b.

The first possibility leads to different boundaries of T and of its complement $S-T$, which may be considered as a topological paradox and contradicts our Axiom 4. The remaining possibility 2 demands that the smallest neighborhoods satisfy the classical Axiom T_0. This is exactly what we wanted to demonstrate.

In the next chapters we will describe some algorithms in digital geometry that are based on the theory of abstract cell complex (AC complexes). In an AC complex the neighborhoods are defined by means of an *antisymmetric* bounding relation. Therefore the boundaries in AC complexes are thin and they satisfy the classical axioms. The author hopes that this will satisfy the desire of a reader of the first category mentioned at the beginning of Introduction. As to a reader of the second category, we have demonstrated that an attempt to "invent" a new "obviously true" set of axioms leads to no new concept of a topological space: the set of the new axioms has turned out to be equivalent to that of classical axioms. We suppose that this will be the case of all other sets of axioms that are in accordance with our "healthy" understanding of topology.

As to the readers of the third category, we hope that they will be convinced and satisfied after having read Part II "The Algorithms".

Part I – Theoretical Foundations

2 Short Remarks on Cell Complexes

We presume that the reader is acquainted with the theory of AC complexes. To make the reading of the paper easier we have summarized the most important definitions in the Appendix. For more details we refer to [2, 5, 8].

3 Data Structures

In this Section we shall describe some data structures used in algorithms for solving topological and geometrical problems when using AC complexes.

3.1 The standard raster

Two- and three-dimensional images are usually stored in a computer in arrays of the corresponding dimension. Each element of the array contains either a gray value, or a color, or a density. This data structure is called the *standard raster*. It is not designed for topological calculations, nevertheless, it is possible to perform topological calculations without changing the data structure. For example, it is possible to trace and encode the frontier of a region in a two-dimensional image in spite of the apparent difficulty that the frontier according to Definition FR (see Appendix) consists of 0- and 1-cells, however, the raster contains only pixels which *must* be interpreted as 2-cells. The reason is that a pixel is a carrier of an optical feature that is proportional to certain elementary area. Thus pixels, which are the 2-cells, must correspond to elementary areas rather than 0- or 1-cells whose area is zero. On the same reason voxels must correspond to 3-cells.

When considering an image as an AC complex one must admit that a 2-dimensional image contains besides the pixels also cells of lower dimension 0 and 1. This often arises objections since we can see on a computer display only the pixels, i.e. the cells of the highest dimension. However, this fact is not important: really, one should think about the rendering of a 3-dimensional object that is a set of voxels. What we see are not the 3-dimensional voxels but rather their 2-dimensional facets and 1-dimensional edges. This fact does not prevent us from working with voxels which we do not see on the display. All these are peculiarities of the human viewing system rather than that of the objects. By the way, it is no problem to make, if desired, the lower dimensional cells visible on a display [2].

The tracing in the standard raster is possible because the concept of an AC complex is the *way of thinking* about topological properties of digitized images rather than a way of encoding them. Let us explain this idea for the case of tracing frontiers.

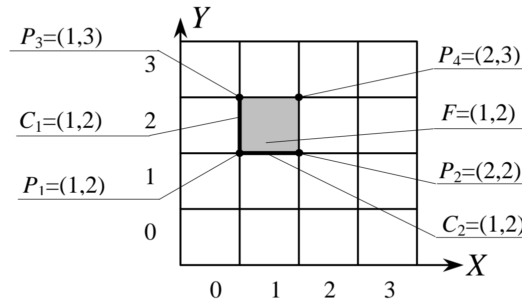


Fig. 3.1. Non-topological coordinates of cells of lower dimensions

We think of a two-dimensional (2D) image as of a 2D Cartesian complex (Appendix) containing cells of dimensions form 0 to 2. The 2-cells (pixels) have in the standard raster coordinates which, unlike to topological coordinates of a pixel being always odd (Appendix and Fig. A.1), may take in the standard raster *any integer values*, both odd and even. Pixels are explicitly represented in the raster, however, the 0- and 1-cells are present implicitly.

Coordinate Assignment Rule: Each pixel F of a 2D image gets one 0-cell assigned to it as its "own" cell. This is the 0-cell lying in the corner of F which is the

nearest to the origin of the coordinates (P_1 in Fig. 3.1). Also two 1-cells incident to F and to P_1 are declared to be own cells of F (C_1 and C_2 in Fig. 3.1). Thus each pixel gets three own cells of lower dimensions. *All own cells of F get the same coordinates as F*. They can be distinguished by their type.

In the three-dimensional case each voxel gets seven own cells of lower dimensions which are arranged similarly. These seven cells get the same coordinates as the corresponding voxel.

Unfortunately, some cells in the boundary of the raster remain without an "owner". In most applications this is of no importance. Otherwise the raster must be correspondingly enlarged.

According to the above rule, it is not difficult to calculate the coordinates of all pixels incident to a given point. This coordinates are used to get the gray values of the pixels from the array containing the image. Depending on these gray values the *virtual* tracing point P moves for one position to the next. The movement is represented by the changing coordinates of the virtual point. The details of the tracing algorithm are described in Section 4.

The majority of low level topological problems in image processing may be solved in a similar way, i.e. without representing cells of lower dimension as elements of some multidimensional array. A typical exception is the problem of filling the interior of a region defined by its frontier (Section 8). The solution is simpler when using two array elements per pixel: one for the pixel itself and one more for its own vertical crack (1-cell). The solution consists in reading the description of the frontier (e.g. its crack code), labeling all vertical cracks of the frontier in the array and in counting the labeled cracks in each row (starting with 0) and filling the pixels between the crack with an even count $2 \cdot i$ and the next crack (with the count $2 \cdot i+1$). The details of this algorithm are described in Section 8

Even more complicated topological problems may be solved by means of the standard raster. For example, when tracing surfaces (Section 7) or producing skeletons (Section 11) the so called simple pixels must be recognized. It is easier to correctly recognize all simple pixels if *cells of all dimensions* of the region under consideration are labeled. To perform this in a standard raster, it is possible to assign a bit of a raster element representing the pixel F to each own cell of F. For example, suppose that one byte of a two-dimensional array is assigned to each pixel of the image shown in Fig. 3.1. Consider the pixel F with coordinates (1, 2) and the byte assigned to it. The bit 0 of the byte may be assigned to the 0-cell P_1, the bit 1 to the 1-cell C_1, the bit 2 to the 1-cell C_2. The remaining bits may be assigned to F itself. Similar assignments are also possible in the 3D case.

As we see, there is no necessity to allocate memory space for each cell of a complex, which would demand four times more memory space than that needed for pixels only, or eight times more than that needed for the voxels in the 3D case.

3.2 The Topological Raster

To explicitly represent an image as a Cartesian AC complex the so called *topological raster* [8] must be used. It is again a 2- or 3-dimensional array in which a memory element is reserved for each cell of each dimension.

In a topological raster *each cell has different coordinates* which simultaneously are the indices of the array. Each coordinate axis is a topological line (Definition TL, Appendix). The 0-cells of the axis have even coordinates, the 1-cells have odd coordinates (Fig. A1, Appendix). The dimension and the orientation (if defined) of any cell may be calculated from its topological coordinates. The dimension of any cell is the *number* of its odd coordinates, the orientation is specified by indicating *which* of the coordinates are odd. For example, the cell C_1 in Fig. A.1 has *one* odd coordinate and this is its *X*-coordinate. Thus it is a *one*-dimensional cell oriented along the *X*-axis. The 2-cell *F* has two and the 0-cell *P* has no odd coordinates. In a three-dimensional complex the orientation of the 2-cells may be specified in a similar way: if the *i*th coordinate of a 2-cell *F* is the only even one then the normal to *F* is parallel to the *i*th coordinate axis.

The advantages of the topological raster are shaded by its drawback that it demands 2^n times more memory space than the standard raster. Also the time needed to process it is correspondingly greater. Therefore it is reasonable to use the topological raster to save images only for not too large images or for research purposes where the processing time is not so important.

The best way to use the topological coordinates consists in the following. When thinking about the problem to be solved one should use the topological coordinates. Also a program which must solve a topological or geometrical problem in which notions like the connectivity, boundary, incidence etc. are involved should work with topological coordinates. Only the storage of values assigned to the ground cells, i.e. to pixels or voxels, should be performed in the standard raster to save memory space. This is possible since the cells of lower dimension do not carry values like color, gray value or density. They only carry indices of subsets to which they belong, i.e. foreground or background. These indices may be specified by means of predefined rules called membership rules [2].

For example, to compute the connected components of a binary 2D image the rule may be formulated which says that each cell of dimension 0 or 1, which is incident to a foreground pixel, belongs also to the foreground. There are relatively seldom problems the solution of which demands that certain values must be assigned to cells of lower dimensions and then evaluated during the computation. For example, when tracing the boundaries of all connected components of a binary 2D image it is necessary to label the cells of the boundaries which are already processed. Otherwise one and the same component may be found many times. This, however, does not mean that one must label all cells of a boundary. If the process of finding a new component looks for vertical 1-cells of a boundary while comparing the values of two adjacent pixels in a row then it is sufficient to label *only the vertical cracks* of the boundary being processed. It is sufficient to have one bit of memory to save such a label. This bit may be located in the byte of the standard raster, which byte is assigned to the pixel lying in the same row as the crack to be labeled just to the right from the crack. In this case 7 bits remain for the gray values of the pixels which is sufficient in the most cases.

It is also possible to have another array for the cracks along that for the pixels. In this case one needs two times more memory space than for the pixels only. However, it is still more economical than using a topological raster which needs 4 times more

memory space. We shall show more examples of using the standard raster while working with cells of lower dimensions in the algorithms of Part II.

Part II – The Algorithms

We describe here some algorithms for computing topological features of subsets in 2D and 3D digitized images. Since the programming languages of the C-family are now more popular than that of the PASCAL-family, we use here a pseudo-code that resembles the C-language.

4 Boundary tracing in 2D images

Boundary tracing becomes extremely simple when thinking of a 2D image as of a 2D complex. The main idea of the algorithm consists in the following: at each boundary point (0-cell) *P* (Fig. 4.1) find the next boundary crack *C* incident to *P* and make a step along the crack to the next boundary point. Repeat this procedure until the starting point is reached again. Starting points of all boundary components must be found during an exhaustive search through the whole image. The following subroutine `Trace()` is called each time when a not yet visited boundary point of a region is found.

To avoid calling `Trace()` more than once for one and the same foreground component vertical cracks must be labeled (e.g. in a bit of `Image[]`) as "already visited". `Trace()` follows the boundary of one foreground region while starting and stopping at the given point `(x,y)`. Points and cracks are present only implicitly (see Section 3). `Trace()` starts always in the direction of the positive Y-axis.

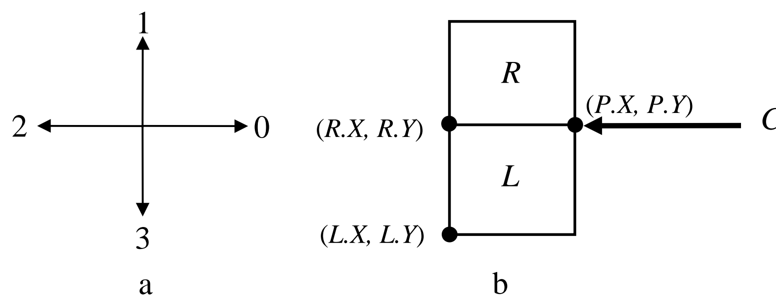


Fig. 4.1. The four directions (a) and the right and left pixels (b)

After each move along a boundary crack *C* the values of *only two pixels R* and *L* of SON(*P*) of the end point *P* of *C* must be tested since the values of the other two pixels of SON(*P*) have been already tested during the previous move. Note that the cracks in a 2D Cartesian complex have only 4 different directions. Therefore the variable "direction" takes the values from 0 to 3 (Fig. 4.1a).

The point corresponding to the corner of a pixel, which is the nearest to the coordinate origin, has the same coordinates as the pixel. For a detailed description of this algorithm see [4].

The pseudo-code of `Trace()`: `Image[NX, NY]` is a 2D array (standard raster) whose elements contain gray values or colors. The variables `P, R, L` and the

elements of the arrays `right[4]`, `left[4]` and `step[4]` are structures each representing a 2D vector with integer coordinates, e.g. `P.X` and `P.Y`. The operation "+" stands for vector addition. Text after // is a comment.

```
void Trace(int x, int y, char image[])
{ P.X=x; P.Y=y; direction=3;
  do
  { R=P+right[direction];   // R is the "right" pixel
    L=P+left[direction];     // L is the "left" pixel
    if (image[R]==foreground)
      direction=(direction+3) MOD 4; // right turn
    else
      if (image[L]==background)
        direction=(direction+1) MOD 4; // left turn
    P=P+step[direction]; //a move in the new direction
  } while( P.X!=x || P.Y!=y);
} // end Trace
```

This simple algorithm is used in any program for the analysis of boundaries in 2D images, e.g. in segmenting the boundary into digital straight segment, in the polygonal approximation of boundaries, in calculating the curvature etc.

5 Segmentation of Digital Curves into Longest DSSs

5.1 Theoretical Preliminaries

Definition HS: A *digital half-space* is a region (Definition RG, Appendix) containing all ground cells of the space, whose coordinates satisfy a linear inequality. A *digital half-plane* is a half-space of a two-dimensional space.

Definition DSS: A *digital straight line segment* (DSS) is any connected subset of the frontier of a digital half-plane.

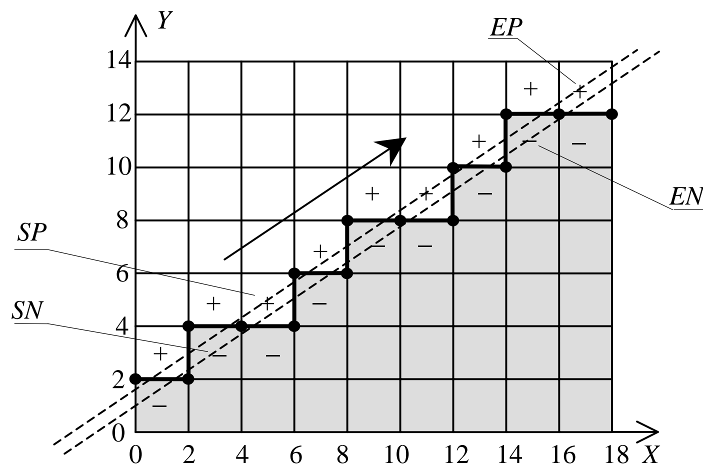


Fig. 5.1. Examples of a half-plane and of a DSS in topological coordinates

Fig. 5.1 shows an example of the half-plane defined by the inequality $2 \cdot x - 3 \cdot y + 3 \geq 0$. All pixels of the half-plane are represented by shaded squares.

We suggest to consider two kinds of digital curves: *visual curves* as sequences of pixels and *boundary curves* as sequences of alternating points and cracks. We consider DSSs as *boundary curves* while in the most publication (e.g. in [14]) they are considered as visual curves, i.e. as sequences of pixels.

Since in practice DSSs are mostly used in image analysis rather than in computer graphics, considering them as boundary curves is more adequate to the demands of applications as this will be shown below in Section 5.3.

5.2 Most Important Properties of a DSS

To investigate the properties of DSSs it is easier and more comprehensible to consider at first the 2-cells (pixels) incident to the cells of a DSS rather than the cells of the DSS themselves. Consider a digital curve K in a two-dimensional space. It is possible to assign an orientation to K and thus to the 1-cells of K. Suppose that K does not intersect the boundary of the space. Then each 1-cell C of K is incident to exactly two pixels. One of them lies to the positive side and the other to the negative side of the ordered pair of the end points of C. (The cell c lies to the positive side of the ordered pair (a, b) if the rotation from b to c about a is in the mathematically positive direction, as e.g. the rotation from the positive X-axis to the positive Y-axis).

Definition PP: A pixel P which is incident to an oriented crack C of the curve K and lies to the positive side of the ordered pair of the end points of C is called a *positive pixel* of K. Similarly, the incident pixel lying to the negative side of the ordered pair of the end points of C is called a *negative pixel* of K.

In Fig. 5.1 the positive pixels are labeled by "+" and the negative ones by "−".

The set of all positive pixels of K will be called the *positive pixel set* of K and denoted by $SP(K)$. The set of all negative pixels of K will be called the *negative pixel set* of K and denoted by $SN(K)$.

If K is a DSS then $SP(K)$ lies in a half-plane while $SN(K)$ lies in its complement. Therefore there is a linear form $H(x, y)$ such that $H(x, y) \geq 0$ for all positive pixels of K and $H(x, y) < 0$ for the negative ones. We shall call $H(x, y)$ the *separating linear form* of the DSS.

The properties of a DSS which we are interested in and which are important for the recognition of a DSS are known from the literature during many years. The majority of the publications consider *visual lines in standard coordinates*. We repeat here the properties necessary for the recognition of DSS and reformulate them for *boundary lines in topological coordinates*. The necessary proofs are to be found in [12]. The most part of our results is applicable for both standard and topological coordinates due to introducing a parameter e which is the minimum distance between two pixels: e is equal to 1 in standard coordinates and to 2 in topological ones.

Definition SSF: The separating linear form $H(x, y) = a \cdot x + b \cdot y + r$ of a DSS D is called the *standard separating form* (SSF) of D if it satisfies the following conditions:

$H(x, y) \geq 0$ for all positive pixels of D, $H(x, y) < 0$ for all negative ones;

The coefficients of $H(x, y)$ are integers while a and b are mutually prime;

There are either at least two positive pixels P_1 and P_2 of D at which $H(x, y)$ takes its minimum value *with respect to* all positive pixels of D or at least two negative

pixels N_1 and N_2 of D at which $H(x, y)$ takes its maximum value *with respect to* all negative pixels of D.

Definition BS: The set of positive pixels of a DSS D at which the SSF of D takes its minimum value with respect to all positive pixels of D is called the *positive base* of D. Similarly, the set of negative pixels of a DSS D at which the SSF of D takes its maximum value with respect to all negative pixels of D is called the *negative base* of D.

The pixel of the base B, which is the nearest to the starting point (respectively, end point) of the oriented DSS is called the *starting pixel* (respectively, the *end pixel*) of B. In Fig. 5.1 the starting pixel of the positive base is denoted by SP, its end pixel is denoted by EP. Similarly, the starting and the end pixel of the negative base are denoted by SN and EN.

It is well-known and proved in [12] that a DSS contains cracks of at most two different directions and that the values of $H(.)$ satisfy the inequalities:

$$0 \le H(x, y) \le e \cdot (\max(|a|,|b|)-1) \quad \text{for positive pixels;}$$

$$-e \cdot \max(|a|,|b|) \le H(x, y) \le -e \quad \text{for negative ones.}$$

Let S and E be the starting and the end pixel of those base of the DSS, which contains more than one pixel. Then the vector parallel to $E-S$ whose components are mutually prime is called the *base vector* of the DSS. If both bases contain a single pixel, which is only the case, when the DSS consists of a single crack, then the base vector is a unit vector parallel to that crack.

The problem most important for applications is that of segmenting a given digital curve into as long as possible DSSs. The author has developed an algorithm [3] which starts with the first two cracks of the curve (two adjacent cracks with the point between them compose always a DSS) and then tests the following cracks one crack after another whether the sequence of cracks is still a DSS. Section 5.3 describes a slightly modified more comprehensible version of the algorithm. The idea is as follows.

The first two cracks uniquely specify the bases and the SSF $H(V)$ of the DSS. The algorithm checks whether the direction of the next crack C is allowed. If not then C does not belong to the actual DSS which ends at the starting point of C. Otherwise the values $H(P)$ and $H(N)$ of the SSF for the two pixels P and N incident to C must be checked. If both $H(P)$ and $H(N)$ are in the allowed intervals then C belongs to the actual DSS. If one of these values is at the boundary of the allowed interval then the corresponding pixel P or N belongs to the corresponding base which must be prolonged until P or N. If one of the values $H(P)$ and $H(N)$ deviates from the boundary of the allowed interval by the value of e (e=1 for standard coordinates and e=2 for topological coordinates) then the coefficients of $H(V)$ must be redefined: the sequence of cracks is a DSS slightly different from the actual DSS. If one of the values $H(P)$ and $H(N)$ deviates from the boundary of the allowed interval by a value greater than e then there is no DSS containing C and all previous cracks. The actual DSS ends at the starting point of C.

The author has proved the correctness of the algorithm [12]. The proofs of all necessary lemmas and theorems take about 10 pages and cannot be repeated here.

5.3 The Algorithm "SubdivideInDSS"

In this section we describe an algorithm which traces a given digital curve *CV* in a 2D image, e.g. the boundary of a region, and subdivides the curve into the longest possible DSSs. The output of the algorithm is a list of the coordinates of the endpoints of the DSSs.

Tracing the curve: Call the subroutine *Ini* (see below). Choose an arbitrary crack *SC* of *CV* , choose its orientation while specifying its direction *dir* as one of the numbers 0 to 3 as shown in Fig. 4.1a. *SC* is the starting crack. Save the coordinates of the starting point of *SC* as the starting point of the first DSS. Set the running crack *C*=*SC* and start the tracing loop. The loop runs through all cracks of *CV*. For each new location of the running crack *C* call the subroutine *Reco*(*C*, *dir*) (see below) where *dir* is the direction of the oriented crack *C*. If the return value of *Reco* is zero go over to the next crack. Otherwise save the starting point of the running crack *C* as the end point of the current DSS and start the recognition of the next DSS while calling *Ini* and *Reco*(*C*, *dir*) again. Stop when the starting crack *SC* is reached again. The curve *CV* is subdivided into as long as possible DSSs by the saved points.

Subroutine *Ini*: Set the crack counter *CC* to zero and the two prohibited directions *Proh*1 and *Proh*2 to −1 as "unknown".

Subroutine *Reco*: It gets as parameters the coordinates of the running crack *C* and its direction *dir*. The subroutine performs as follows.

As the first step it calculates the coordinates of the positive *P* and the negative *N* pixels incident to *C* while considering the value of *dir*.

If the counter *CC*=0 then set the first prohibited direction *Proh*1 as opposite to *dir*. Set the starting pixel *StartP* and the end pixel *EndP* of the positive base both equal to *P*. Similarly set *StartN* and *EndN* equal to *N*. Set the base vector (*b*, −*a*) equal to the unit vector with the direction *dir*. Increment *CC*; return 0.

If *CC*≠0 then, independently upon the value of CC, test whether *dir* is equal to one of the prohibited directions. If this is the case then return 1: the actual crack *C* does not belong to the running DSS. If *dir* is not prohibited and *Proh*2 is still unknown then set *Proh*2 as opposite to *dir*.

If *CC*=1 then
BEGIN

Set *EndP*=*P* and *EndN*=*N* and increment *CC*. If *Proh*2 is still unknown return 0: the first two cracks have the same direction, the parameters *a* and *b* remain unchanged. Otherwise calculate the coefficients *a*, *b*, *r* of the separating linear form $H(x, y)=a\cdot(x-StartP.x)+b\cdot(y-StartP.y)$ so that $H(x, y)=0$ for the positive pixels and $H(x, y)=-e$ for the negative ones. Concretely: if the endpoints of the positive base coincide then

$$a=-(EndN.y-StartN.y)/e;$$
$$b=(EndN.x-StartN.x)/e;$$

otherwise (5.3.1)

$$a=-(EndP.y-StartP.y)/e;$$
$$b=(EndP.x-StartP.x)/e;$$

Return 0.

END IF

For all subsequent cracks ($CC>1$) calculate the values *HP* and *HN* of the separating form for *P* and for *N*: $HP=H(P)$ and $HN=H(N)$.

The following steps of the Algorithm are the decisive ones:

If $HP<-e$ or $HN>0$ then return 1: the actual crack *C* does not belong to the running DSS.

If $HP=0$ then set *EndP*=*P*: *P* lies on the positive base which must be prolonged.

Otherwise, if $HP=-e$ then redefine both bases while setting *EndP*=*P*; *StartN*=*EndN* and redefine the parameters *a* and *b* of $H(x, y)$ according to the redefined bases.

If $HN=-e$ then set *EndN*=*N*: *N* lies on the negative base which must be prolonged.

Otherwise, if $HN=0$ then redefine both bases while setting *EndN*=*N*; *StartP*=*EndP* and redefine the parameters *a* and *b* of $H(x, y)$ according to the redefined bases.

return 0.

End of Subroutine *Reco*.

The pseudo-code of Reco:

```
int CRecoDSS::Reco(CPoint Crack, int dir)
{ P=Crack+ToPos[dir];     // The Array "ToPos[4]" contains vectors pointing
                          // from a crack to its positive pixel P
  N=Crack-ToPos[dir];     // N is the negative pixel of Crack
  if (CC==0)              // "CC" is the number of tested cracks
  { Prohibit1=Opposite(dir);  Prohibit2=-1; CC=1;
    StartP=EndP=P;    StartN=EndN=N;
    a=−Param[dir].y;   b=Param[dir].x;  // a unit vector along "dir"
    return 0;
  }
  if (dir==Prohibit1 || dir==Prohibit2) return 1;
  if (dir!=Opposite(Prohibit1) && Prohibit2==-1)
            Prohibit2=Opposite(dir);
if (CC==1)
  { EndP=P; EndN=N; CC=2;
    if (Prohibit2==-1) return 0;  // only one direction
    if (EndP==StartP)
    { a=−(EndN.y-StartN.y)/e;  b=(EndN.x-StartN.x)/e;
    }
    else
    { a=−(EndP.y-StartP.y)/e;   b=(EndP.x-StartP.x)/e;
    }
    return 0;        // any two allowed cracks compose a DSS
  }
  int HP=GetH(P);  int HN=GetH(N);  //the values of the SSF "H"
  if (HP<-e || HN>0) return HP;      // not a DSS
  if (HP==0)  EndP=P;
  else
    if (HP==-e)
    { EndP=P; StartN=EndN;
```

```
      a=-(EndP.y-StartP.y)/e; b=(EndP.x-StartP.x)/e;
    }
  if (HN==-e) EndN=N;
  else
    if (HN==0)
    { EndN=N; StartP=EndP;
      a=-(EndN.y-StartN.y)/e; b=(EndN.x-StartN.x)/e;
    }
  return 0;
} //************************** end Reco ******************************
```

The recognition of DSS is one of the fastest and most economical methods of encoding geometrical objects. For example, it is possible to encode the boundary of a region as a sequence of DSSs while saving the coordinates of a single point and four integer parameters for each DSS [6]. The parameters *exactly* specify the location of the DSS in the image thus enabling the *exact* reconstruction of the region. The author has developed an economical code which needs on the average 2.3 bytes per one DSS. When encoding a quantified gray value image a compression rate of 3.1 was reached [6]. The method is very fast: for example, it encodes a binary image of 640×480 pixels containing about 30 disk-shaped objects in 20 ms on a PC with a processor of 700 MHz. In that time also the recognition of the disks and estimating their locations and diameters was performed.

6 Recognition of Digital Plane Patches (DPP)

The notion of a digital plane patch (or segment) is well known from the literature. [1, 15]. The DPPs were mostly considered as sets of voxels, i.e. as three-dimensional objects ("thick DPPs"). However, it is more appropriate to consider a DPP as a subset of a surface, i.e. as a two-dimensional object. Our approach based on cell complexes makes it possible.

Definition DPP: A connected subset of the frontier of a three-dimensional half-space (Definition HS, Section 5.1) is called a *digital plane patch* (DPP).

A DPP contains no voxels (as any frontier in a 3D space does). It contains only 2-cells called facets, 1-cells called cracks and 0-cells called points.

6.1 The Problem of the Segmentation of Surfaces into DPPs

This problem is of great practical importance since its solution promises an economical and precise encoding of 3D scenes.

The Problem Statement:

Given: a surface in a 3D space, e.g. the boundary of a connected subset.

Find: the minimum number of DPPs representing the surface in such a way that it is possible to reconstruct the surface, at least with a predefined precision, from the code of the DPPs.

The problem consists of two partial problems:

1. *Recognition:* Given a set of facets decide whether it is a DPP or not.
2. *Choice*: Given a surface S and a subset of facets, which is known to be a DPP, decide which facet of S should be appended to the subset to achieve that the number of the DPPs in the segmentation of S be minimal.

6.2 The Partial Problem of the Recognition of a DPP

Similarly as in the case of the DSS (Section 5), a surface S specifies two sets of voxels incident to the facets of S: a positive and a negative set. The voxels of the positive set lie outside of the body whose boundary is S, that of the negative set lie inside. If a subset T^2 of facets of S is a DPP then it lies in the frontier of a half-space whose voxels satisfy a linear inequality. The inequality separates the positive voxels of T^2 from its negative voxels.

To decide whether T^2 is a DPP it is sufficient to solve the following system of $2 \cdot N$ linear inequalities in the components H_k, k=1, 2, 3; of a 3D vector $\boldsymbol{H}$ and a scalar value C where N is the number of facets in T^2.

$$\sum_k H_k \cdot V_k^+(F_i) - C \geq 0;$$

$$\text{with } F_i \in T^2;\ i=1...N; \qquad \text{(UV)}$$

$$\sum_k H_k \cdot V_k^-(F_i) - C < 0;$$

$V_k^+(F_i)$ stays for the kth topological coordinate of the positive voxel incident to the face F_i. Similarly, $V_k^-(F_i)$ is the kth topological coordinate of the negative voxel incident to F_i. The vector $\boldsymbol{H}$ is the normal to a plane separating all positive voxels from the negative ones, while C specifies the distance of the plane from the coordinate origin.

It is possible to solve the problem by a fast method similar to that of recognizing a DSS (Section 5). The corresponding algorithm and the related theory are, however, rather complicated. Their presentation here is impossible because of the page limit. We describe a rather simple algorithm [13] whose only drawback is its low speed: it is an O(N^2) algorithm. Nevertheless, the algorithm is well suited for research purposes.

The algorithm solves the following problem:

Given are two sets M^+ and M^- of points in an n-dimensional space.

Find an (n–1)-dimensional hyperplane HP separating the sets.

The solution: HP is specified by two vectors A^+ and A^- as the middle perpendicular to the line segment (A^+, A^-). Let $Dist(P)$ be the signed distance of the point P to HP.

Fig. 6.1 shows an example of two point sets in a 2D space.

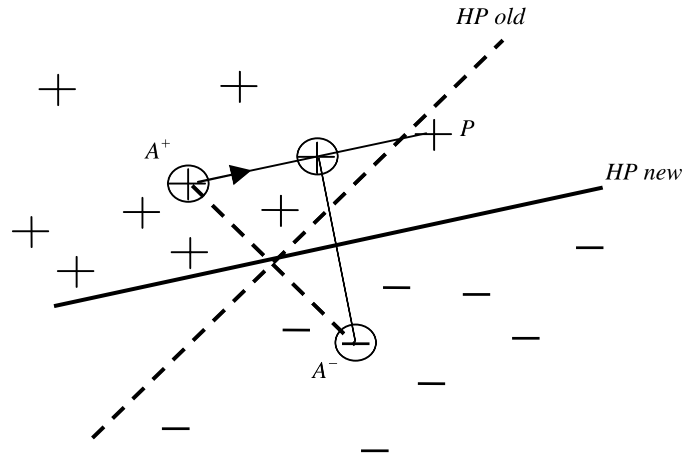


Fig. 6.1. An example of the correction of the separating plane

The Algorithm:

1. Set A^+ equal to an arbitrary point from M^+ and A^- equal to an arbitrary point from M^-. Carry out a sequence of the following iterations.
2. During each iteration test all points P from M^+ and M^- as follows:
   If the point $P \in M^+$ lies on the wrong side of *HP* which means $Dist(P)<0$, then set:
   A^+:= *Foot of the perpendicular from* A^- *to the segment* (A^+, P).
   If $P \in M^-$ and $Dist(P)>0$, then set:
   A^-:= *Foot of the perpendicular from* A^+ *to the segment* (A^-, P).
3. If there is no point on the wrong side of *HP* stop the Algorithm. The separating hyperplane is the middle perpendicular to the line segment (A^+, A^-).
4. If the distance between A^+ and A^- is less than a predefined threshold then there exists no separating hyperplane; the convex hulls of M^+ and M^- intersect.

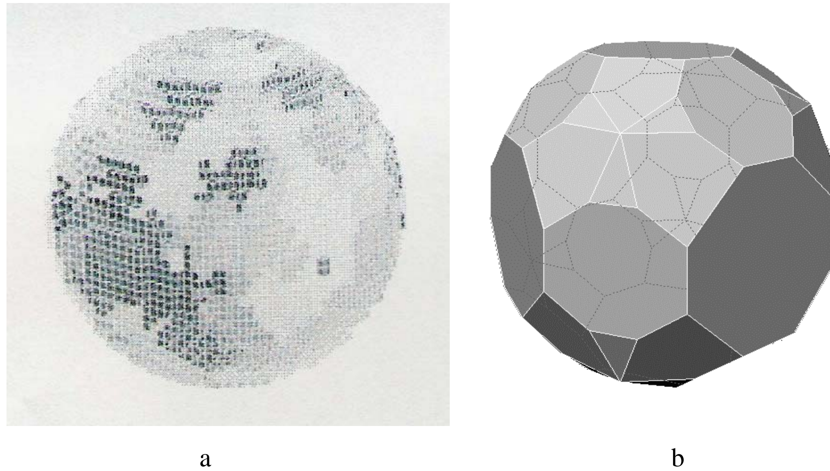
Fig. 6.1 shows an example of separating two point sets in a 2D space. The point sets are represented by "+" and "–" signs, the vectors A^+ and A^- by encircled signs; the old and the new separation plane by a dashed and a solid line. The point P lies on the wrong side of the old plane. It is connected with the old vector A^+ and a perpendicular has been dropped from A^- onto the segment (A^+, P). The foot of the perpendicular is the new vector A^+.

We have applied this method to recognize DPPs while using the set of the positive voxels V^+ as M^+ and that of the negative voxels V^- as M^-.

6.3 The Partial Problem of the "Choice"

There is no efficient method for the solution of the partial problem "Choice" known until now. It should be mentioned that this problem is much more difficult in the case of DPPs as in the case of DSSs. Really, in the latter case there are exactly two possibilities to continue a partially recognized DSS: forward or backward along the digital curve. But in the case of a DPP there are as many possibilities to continue as

the number of facets adjacent to a partially recognized DPP. There is no known criterion to decide which of them should be preferred. When choosing the next facet arbitrarily then the found DPPs look chaotic even for surfaces of regular polyhedrons (Fig. 6.2a).



a b

Fig. 6.2. Examples of segmenting the boundary of a digital ball into DPPs by arbitrarily choosing the next facet (a) and by computing its convex hull (b)

7 Tracing Surfaces in 3D

The algorithm [7] presented here traces a surface *S* of a three-dimensional body in a way similar to that of peeling a potato: the facets of *S* are visited one after another composing a continuous sequence. Each facet is encoded by one byte. If the genus of *S* is zero then each facet is contained in the encoded sequence only once. The code of a surface of a greater genus contains a small number of facets many times which makes the code a little longer. In any case it is possible to reconstruct the surface and the body from the code exactly.

To explain the algorithm we need the following notions from the theory of AC complexes:

Definition OF: The *open frontier* Of(L, S) of a subcomplex L of a complex S *relative to S* is the subcomplex of S containing all cells C of S whose closure Cl(C, S) contains both cells of L as well as cells of the complement S–L.

Definition SI: A facet F of S is called *simple relative to the subcomplex L* if $F \notin L$, the intersection Cl(F, S)$\cap L$ is connected and Cl(F, S)$\cap$(S–L)$\neq\emptyset$.

The algorithm works as follows: It chooses an arbitrary facet of the surface S as the starting one and labels its closure. Then it traces the open frontier Of(L, S) of the set L of labeled cells, encodes the facets of Of(L, S) (1 byte per facet), and labels the closures of simple facets.

This ensures that L remains homeomorphic to a closed 2-ball (a disk).

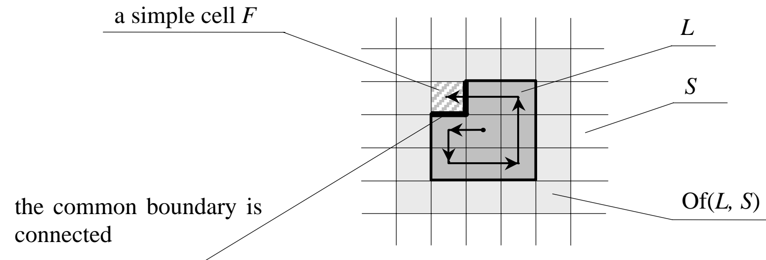


Fig. 7.1. The moves at the beginning of the tracing

The author has proved that if the surface S is homeomorphic to a sphere then the traced sequence is a Hamilton path: each facet is visited exactly once. Otherwise there remain a few non-simple facets which are visited at least twice. Their code elements are attached to the end of the sequence of simple facets. Thus the code sequence is always connected. A verbal description of the algorithm follows. A detailed description and the related proofs may be found in [7].

The Algorithm:

Notations: S is the surface to be traced. $L \subset S$ is the subset of labeled cells; it is homeomorphic to a closed 2-ball. The "rest sequence" is the set of non-simple facets at the stage when all simple facets of S are already labeled. The rest sequence consists of a single facet if the genus of S is zero.

1. Take any facet of S as the starting facet F_0, label its closure and save its coordinates as the starting coordinates of the code. This is the seed of L, i.e. $L=\{F_0\}$. Denote any one crack of the boundary Fr(F_0,S) as C_{old} and find the facet F of S which is incident to C_{old} and adjacent to F_0. Set F_{old} equal to F_0 and the logical variable *REST* to FALSE. *REST* indicates that the tracing of the rest sequence is running.
2. (Start of the main loop) Find the crack C_{new} as the first unlabeled crack of Fr(F,S) encountered during the scanning of Fr(F,S) clockwise while starting with the end point of C_{old}, which is in Fr(L,S). If there is no such crack and F is labeled stop the Algorithm: the encoding of S is finished.
3. If F is simple label its closure.
4. Put the direction of the movement from F_{old} to C_{old} and that of the movement from C_{old} to F into the next byte of the code. If the facet F is non-simple set the corresponding bit in the code (to recognize codes of non-simple facets in the ultimate sequence).
5. If *REST* is TRUE check, whether F is equal to F_{stop} and C_{new} is equal to C_{stop}. (These variables were defined in item 6 of the previous loop). If this is the case stop the Algorithm and analyze the rest sequence to specify the genus of S as explained in [7]. Delete multiple occurrences of facets from the rest sequence.
6. If F is simple set *REST* equal to FALSE; else set F_{stop} equal to F, C_{stop} equal to C_{new} and *REST* equal to TRUE.

7. Set F_{old} equal to F. Find the facet F_{new} of S incident to C_{new} and adjacent to F. Set F equal to F_{new} and C_{old} equal to C_{new}. Go to item 2.

End of the Algorithm.

The algorithm was successfully tested by the author for surfaces of genus up to 5. The bodies were exactly reconstructed. A graduate student of the University of Applied Sciences Berlin [16] has also programmed the algorithm and has made a lot of successful experiments with very complicated bodies of high genus.

8 Filling the Interiors of Surfaces in Multi-Dimensional Images

To test whether an n-cell P of an n-space lies in the interior of a given closed hyper-surface S it is sufficient to count the intersections of S with a ray from P to any point outside the space: iff the count is odd then P is within S. However, it is difficult to distinguish between intersection and tangency (Fig. 8.1 a and b).
The solution becomes easy if the surface is given as one or many (n–1)-dimensional manifolds in an n-dimensional Cartesian AC complex and the "ray" is a sequence of alternating n- and (n–1)-cells all lying in one row of the raster (Fig. 8.1c).

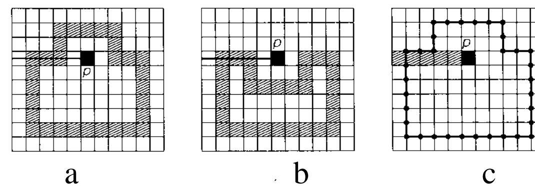


Fig. 8.1. Intersection (a) and tangency (b) are difficult to distinguish in "thick" boundaries; this is easy at boundaries in complexes (c)

In a 2D image the "surface" must be a closed sequence of cracks and points (Fig. 8.1c). Then intersections are only possible at vertical cracks and the problem of distinguishing between intersections and points of tangency does not occur. If one knows which n-cells are within S then one can fill the interior of S by labeling these cells. The method has been successfully implemented for dimensions n=2, 3, 4.

The pseudo-code:

Denote by `F` the current n-cell of the n-dimensional standard raster. Choose a coordinate axis A of the Cartesian space (e.g. A=X in the 2D case). Denote by `C(F)` the (n–1)-cell incident to `F`, whose normal is parallel to A (e.g. the vertical crack incident to `F` in the 2D case). Label all (n–1)-cells of S whose normal is parallel to A. In the 2D case when A=X these are the vertical cracks of S.

```
   for each row R parallel to A do
   { BOOLEAN fill=FALSE;
     for each n-cell F in the row R do
     { if C(F) is labeled then fill=NOT fill;   // inverting fill
       if fill is TRUE then F=foreground;
       else                 F=background;
     }
}
```

9 Component Labeling in an n-Dimensional Space

We consider here the simplest case of a 2D binary image in a standard raster while the algorithm is applicable also to multi-valued and multi-dimensional images in a topological raster. In a standard raster a function must be given which specifies which raster elements are adjacent to each other and thus are connected if they have the same color. In our simple 2D example we use the well-known "(8,4)-adjacency". In the general case the adjacency of the n-cells of an n-dimensional complex must be specified by rules specifying the membership of cells of lower dimensions [2] since an (a, b)-adjacency is not applicable for multi-valued images [5].

In a topological raster the connectivity of two cells is defined by their incidence which in turn is defined by their topological coordinates (Section 3.2).

The Algorithm:
It is expedient to consider a multi-dimensional image as a one-dimensional array *Image*[*N*]. For example, in the 2D case the pixel with coordinates (x, y) may be accessed as *Image*[$y \cdot NX+x$] where NX is the number of pixels in a row. The value $y \cdot NX+x$ is called the *index* of the pixel (x, y).

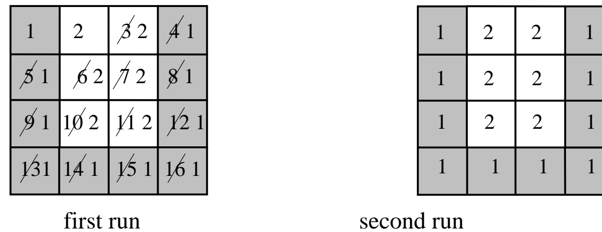


Fig. 9.1. Illustration to the algorithm of component labeling

Given is a binary array `Image[]` of `N` elements and two functions `NumberNeighb(color)` and `Neighb(i,k)`. The first function returns the number of adjacent pixels depending on the color of a given pixel; the second one returns the index of the `k`th neighbor of the `i`th pixel. As the result of the labeling each pixel gets additionally (in another array `Label[]`) the label of the connected component which it belongs to.

The pseudo-code:
Allocate the array `Label[N]` of the same size as `Image[N]`. Each element of `Label` is initialized by its own index:

```
for (i=1; i<N; i++) Label[i]=i; // first loop
for (i=1; i < N; i++)
{ color=Image[i];
  for (j=0; j<NumberNeighb(color); j++)
  { k=Neighb(i, j); //the index of the jth neighbor of i
    if (Image[k]==color) SetEquivalent(i,k,Label);
  }
} // end of the first run
SecondRun(Label,N); // end of the Algorithm
```

Each element of `Label[N]` must have at least $\log_2 N$ bits N being the number of elements in `Image[N]`. The subroutine `SetEquivalent()` makes the preparation for labeling the pixels having the indices `i` and `k` as belonging to one and the same component. For this purpose the subroutine find the "roots" of both pixels `i` and `k`, and the greater root gets the smaller one as its label. The function `Root()` (see below) returns the last value in the sequence of indices where the first index `k` is that of the given pixel, the next one is the value of `Label[k]` etc. until `Label[k]` becomes equal to `k`. The subroutine `SecondRun()` replaces the value of `Label[k]` by the value of a component counter or by the root of `k` depending on whether `Label[k]` is equal to `k` or not.

Pseudo-codes of the subroutines:

```
subroutine SetEquivalent(i,k,Label)
{ if (Root(i,Label)<Root(k,Label))
       Label[Root(k,Label)]=Root(i,Label);
  else Label[Root(i,Label)]=Root(k,Label);
} // end of SetEquivalent

int Root(k, Label)
{ do
  { if (Label[k]==k) return k;
    k=Label[k];
  } while(1);
} // end of Root

subroutine SecondRun(Label,N)
{ count=1;
  for (i=0; i<N; i++)
  { value=Label[i];
    if (value==i)
    { Label[i]=count;  count=count+1;
    }
    else Label[i]=Label[value];
  }
} // end of SecondRun
```

10 Computing the Curvature of Digital Curves

The most of the algorithms for computing the curvature suggested during the last decades have a common drawback: they have a very low precision. The reason is that the precision of calculating the curvature of a curve depends dramatically on the precision of estimating the coordinates of the points of the curve. Coordinates of points in digital images are specified with a precision of about ±0.7 pixel. The author has demonstrated [9] that estimating the curvature with a precision of for example 10% is impossible for digital curves with the curvature radius less than 270 pixels because of not sufficient precision of the coordinates.

We suggest a method of using the gray values in a digital image to essentially increase the precision of estimating the coordinates. The method is applicable for gray value images of objects having an almost constant brightness against a homogeneous background.

In this case the gray values at the boundary of the object contain information about the subpixel position of the boundary.

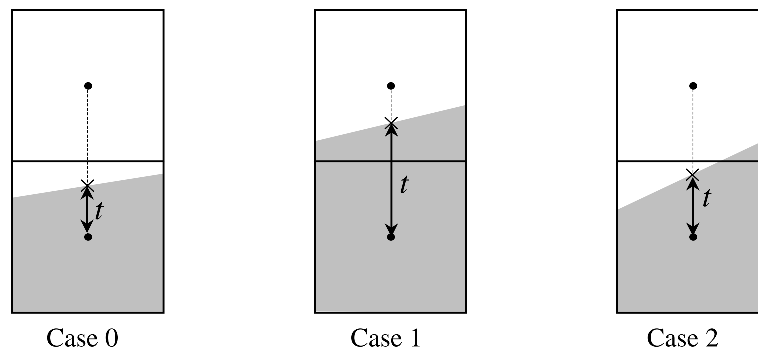


Fig. 10.1. Dependence of the portion of a pixel covered by the object on the subpixel location of its boundary (marked by a cross)

The precision of estimating the coordinates is about 1/100 of a pixel. The curvature may be calculated as the inverse of the radius of a circle through three subpixel points. The optimal distance between the points must be calculated as a function of a coarse estimate of the expected curvature [9].

The precision of estimating the curvature is essentially higher than that of known methods. Fig. 10.2 and 10.3 show an example.

Fig. 10.2. The gray value image of a link of a bicycle chain

The curvature radius of the left and right outer boundary of the object shown in Fig. 10.2 is about 25 pixels. The curvature was estimated with a relative error of about 4% (see Fig. 10.3). When estimating the curvature starting with pixel coordinates without using the gray values the relative error would be according to equation (13) of [9] at least 33%.

Fig. 10.3 shows the curvature of the outer boundary of the object of Fig. 10.2 calculated by our method.

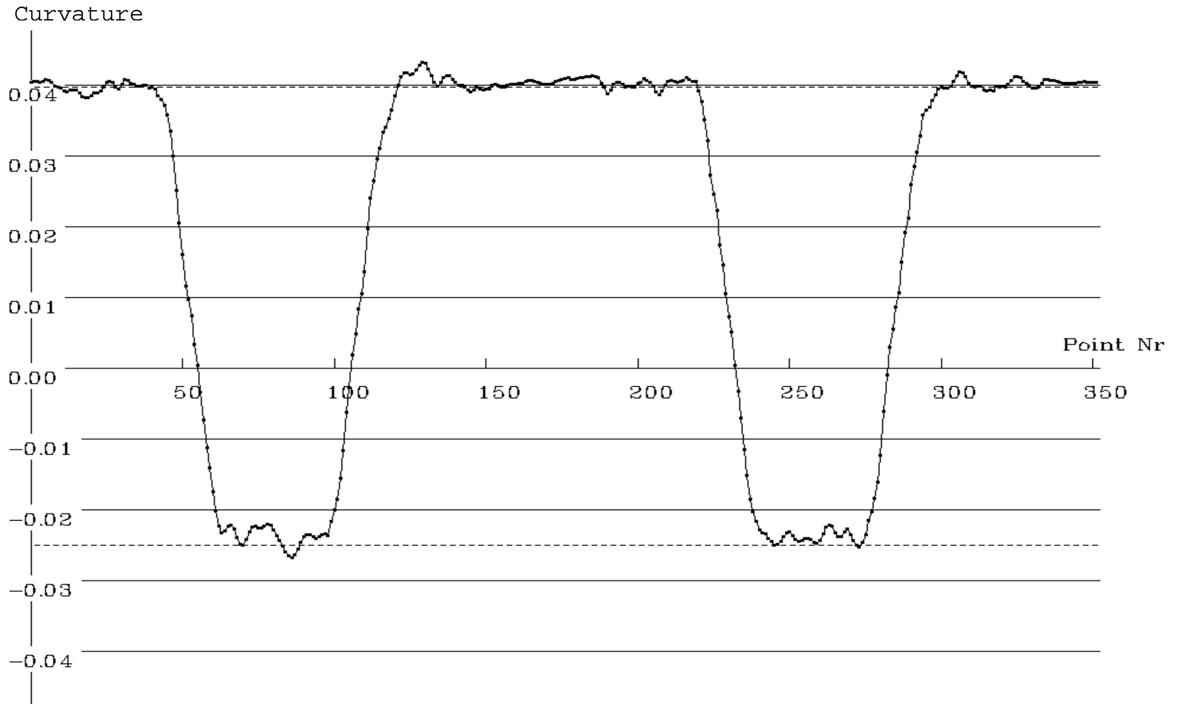


Fig. 10.3. The curvature of the outer boundary of the object of Fig. 10.2; the dashed lines show the true values of the minimum and maximum curvature

The relative error is of about 4%. We hope that Fig. 10.3 gives a general impression of the accuracy of the method as applied to real objects.

11 Skeleton of a Subset in 2D and 3D Images

Definition SK: The skeleton of a given set T of an n-dimensional (n=2,3) image I is a subset $S \subset T$ with the following properties:

a) S has the same number of connected components as T;

b) The number of connected components of $I–S$ is the same as that of $I–T$;

c) Certain singularities of T are retained in S;

d) S is the smallest subset of T having the properties a) to c).

Singularities may be defined e.g. as the "end points" in a 2D image or "borders of layers" in a 3D image etc.

When considering an n-dimensional image (n=2,3) as an AC complex I the problem of the skeletonization consists in finding the condition under which a cell of the foreground T may be reassigned to the background B=$I–T$ without changing the number of the components of both T and B. To derive the condition we need the following notion:

Definition IS: The complex $\mathrm{IS}(C, I)=\mathrm{SON}(C, I)\cup\mathrm{Cl}(C, I)-\{\mathrm{C}\}$ is called the *incidence structure* of the cell C relative to the space I. It is the complex consisting of all cells incident to the cell C excluding C itself [10].

The author has proved that the membership of a cell C of any dimension may be changed between T and B without changing the number of the components of both T and B iff each of the intersections $\mathrm{IS}(C, I)\cap T$ and $\mathrm{IS}(C, I)\cap B$ are not empty and connected. We shall call such cells *IS-simple* relative to the set T. In the rest of this Section we call them "simple".

Thus to calculate the skeleton of a set T one must remove all IS-simple cells of T while reassigning them to B and regarding the singularities.

A well-known difficulty in calculating skeletons is that it is impossible to remove all simple pixels simultaneously without violating the skeleton conditions. However, representing an image as a complex C makes it possible to calculate the skeleton by a procedure which may be either sequential or parallel. It is based on the notion of the *open frontier* (s. Section 7 above). The procedure consist in removing IS-simple non-singular cells of T alternatively from the frontier $\mathrm{Fr}(T, C)$ and from the open frontier $\mathrm{Of}(T, C)$. We present below a simple version of the algorithm for the 2D topological raster.

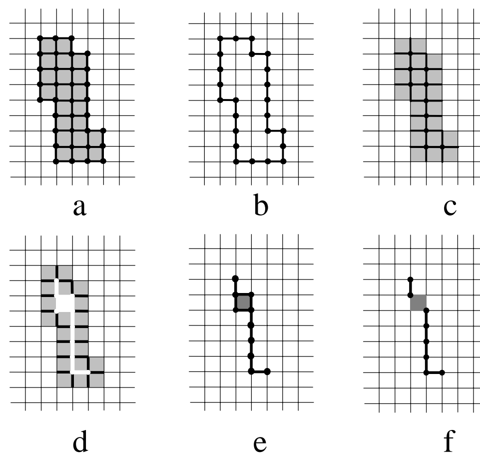


Fig. 11.1. a) a given 2D subcomplex T; b) its frontier Fr; c) the set $T-Fr$: the simple cells of the frontier deleted; d) the open frontier Of of the set $T-Fr$; e) the set $T-Fr-Of$: the simple cells of the open frontier deleted; f) the skeleton

The Algorithm:

Let $I[NX, NY]$ be a 2D array with topological coordinates. The subset T is given by labeling cells of all dimensions of T: $I[x, y] > 0$ iff the cell $(x, y)\in T$. To delete a cell means to set its label $I[x, y]$ to zero. A cell C is *singular* iff it is incident to exactly one labeled cell other than C.

To calculate the skeleton of T run the following loop:

```
do { Scan I and delete all simple and non-singular cells of T∩Fr(T, I);
     CountClose = number of cells deleted during this scan;
     Scan I and delete all simple and non-singular cells of T∩Of(T, I);
     CountOpen = number of cells deleted during this scan;
   } while (CountClose+CountOpen > 0);
  // end of Algorithm
```

Fig. 11.1 above shows an example. The result may be, if desired, easily transformed either to a sequence of pixels or to a one-dimensional complex containing only points and cracks.

In the case of 3D images the algorithm can produce either 1D or 2D skeletons depending upon the choice of the kind of the singularities.

References

(most of the publications by the author are available at "www.kovalevsky.de")

[1] Andres, E.: Le Plan Discret, Colloque "Geometrie Discrete en Imagerie", University of Strasbourg, pp. 45-61, 1993.

[2] Kovalevsky, V.: Finite Topology as Applied to Image Analysis. Computer Vision, Graphics and Image Processing 45 (1989) 141-161

[3] New Definition and Fast Recognition of Digital Straight Segments and Arc*s*, 10th International Conference on Pattern Recognition, Atlantic City, June 17-21, IEEE Press, **vol. II**, pp. 31-34, 1990.

[4] Kovalevsky, V.: Finite Topology and Image Analysis. In: Hawkes, P. (ed.): Advances in Electronics and Electron Physics, Vol. 84. Academic Press (1992) 197-259

[5] Kovalevsky, V.: Digital Geometry Based on the Topology of Abstract Cell Complexes", Proceedings of the Third International Colloquium "Discrete Geometry for Computer Imagery", University of Strasbourg, September 20-21, pp. 259-284, 1993.

[6] Kovalevsky, V.: Applications of Digital Straight Segments to Economical Image Encoding, In: Ahronovitz, E., Fiorio, Ch. (eds), Discrete Geometry for Computer Imagery, Lecture Notes in Computer Science, Vol. 1347, Springer-Verlag, Berlin Heidelberg New York (1997), pp. 51-62

[7] Kovalevsky, V.: A Topological Method of Surface Representation. In: Bertrand, G., Couprie, M., Perroton, L. (eds.): Discrete Geometry for Computer Imagery. Lecture Notes in Computer Science, Vol. 1568. Springer-Verlag, Berlin Heidelberg New York (1999), 118-135

[8] Kovalevsky, V,: Algorithms and Data Structures for Computer Topology. In: Bertrand, G. et all (eds.), Lecture Notes in Computer Science, Vol. 2243: Special issue on Digital and Image Geometry, Springer-Verlag, Berlin Heidelberg New York (2001), pp. 37-58.

[9] Kovalevsky, V,: Curvature in Digital 2D Images, International Journal of Pattern Recognition and Artificial Intelligence, Vol. 15, No. 7, (2001) pp. 1183 - 1200

[10] Kovalevsky, V,: Multidimensional Cell Lists for Investigating 3-Manifolds. Discrete Applied Mathematics, Vol. 125, Issue 1, (2002) pp. 25-43.

[11] Kovalevsky, V.: Axiomatic Digital Topology, to be published, 2004.

[12] Kovalevsky, V.: Recognition of Digital Straight Segments in Cell Complexes, to be published, 2004.

[13] Kozinets, B.N.: An Iteration-Algorithm for Separating the Convex Hulls of two Sets. In Vapnik, V.N. (ed): Lerning Algorithms for Pattern Recognition (in Russian language), Publishing house "Sovetskoe Radio", Moscow, 1973.

[14] Reveillès, J.P.: Structure des Droit Discretes. Journée mathématique et informatique, (1989).

[15] Reveillès, J.P.: Combinatorial Pieces in Digital Lines and Planes. In: Vision geometry III. Proceedings of SPIE, Vol. 2573 (1995) pp. 23-34.

[16] Urbanek, C. : Computer Graphics Tutorials - Visualizing of the Kovalevsky's Algorithm for Surface Presentation, Master thesis, University of Applied Sciences, Berlin, 2003.

Appendix: The Topology of Abstract Cell Complexes

An abstract cell complex (AC complex) is a locally finite topological space (LFS). Elements of this space are called *cells*. As we have seen in Introduction, in a space satisfying the Axioms 1 to 4 which are equivalent to the classical Axioms C1 to C4, neighborhoods must be defined by means of an *antisymmetric* binary relation. It is usual to use in the topology of cell complexes the so called *bounding relation* which is antisymmetric, irreflexive and transitive. Thus it is a partial order.

It is possible either to consider the cells as subsets of an Euclidean space or as some abstract objects having certain properties and relations to each other. In the first case the complex is called Euclidean complex, in the second case it is an abstract cell complex, or AC complex. The author is successfully working with AC complexes and is convinced that considering besides the complex itself also the Euclidean space in which the complex is embedded brings no advantages. The concept of AC complexes opens the exiting possibility to develop digital topology and digital geometry independently of the general topology and of Euclidean geometry.

The most important notions are surely acquainted to the reader from earlier publications (e.g. [2, 5, 8]. Therefore we shall repeat here only the definitions which are necessary to follow the presentation. Other definitions, necessary for certain algorithms are given at the corresponding place.

Definition AC: An *abstract cell complex* (AC complex) $C=(E, B, dim)$ is a set E of abstract elements (cells) provided with an antisymmetric, irreflexive, and transitive binary relation $B \subset E \times E$ called the *bounding relation*, and with a dimension function $dim: E \to I$ from E into the set I of non-negative integers such that $dim(e') < dim(e'')$ for all pairs $(e',e'')\in B$.

The maximum dimension of the cells of an AC complex is called its dimension. We shall mainly consider complexes of dimensions 2 and 3. Their cells with dimension 0 (0-cells) are called *points*, cells of dimension 1 (1-cells) are called *cracks* (edges), cells of dimension 2 (2-cells) are called *pixels* (or facets) and that of dimension 3 are the *voxels*.

If $(e', e'')\in B$ then it is usual to write $e'<e''$ or to say that the cell e' *bounds* the cell e''. Two cells e' and e'' of an AC complex C are called *incident to each other in C* iff either $e'=e''$, or e' bounds e'', or e'' bounds e'. In AC complexes no cell is a subset of another cell, as it is the case in simplicial and Euclidean complexes. Exactly this

property of AC complexes makes it possible to define a topology on the set of abstract cells independently from any Hausdorff space.
The topology of AC complexes with applications to computer imagery has been described in [2]. We recall now a few most important definitions. In what follows we say "complex" for "AC complex".

Definition SC: A *subcomplex S = (E', B', dim')* of a given complex *C = (E, B, dim)* is a complex whose set *E'* is a subset of *E* and the relation *B'* is an intersection of *B* with $E' \times E'$. The dimension *dim'* is equal to *dim* for all cells of *E'*.

Since a subcomplex is uniquely defined by the subset E' it is possible to apply set operations as union, intersection and complement to complexes. We will often say "subset" while meaning "subcomplex".

The *connectivity* in complexes is the *transitive hull of the incidence relation*. It can be shown that the connectivity thus defined corresponds to classical connectivity.

Definition OP: A subset *OS* of cells of a subcomplex *S* of a complex *C* is called *open in S* if it contains all cells of *S* bounded by cells of *OS*. An *n*-cell c^n of an *n*-dimensional complex C^n is an open subset of C^n since c^n bounds no cells of C^n.

Definition SON: The smallest subset of a set *S* which contains a given cell *c* of *S* and is open in *S* is called the *smallest (open) neighborhood* of *c* relative to *S* and is denoted by SON(*c*, *S*).

The word "open" in "smallest open neighborhood" may be dropped since the smallest neighborhood is always open, however, we prefer to retain the notation "SON" since it has been used in many publications by the author.

Definition CL: The smallest subset of a set *S* which contains a given cell *c* of *S* and is closed in *S* is called the *closure* of *c* relative to *S* and is denoted by Cl(*c*, *S*).

Definition FR: The *frontier* Fr(*S*, *C*) of a subcomplex *S* of a complex *C relative to C* is the subcomplex of *C* containing all cells *c* of *C* whose SON(*c*, *C*) contains both cells of *S* as well as cells of the complement *C*–*S*.

Illustrations to AC complexes, SONs and closures of cells of different dimensions may be found in [2, 4, 10].

Definition OF: The *open frontier* Of(*S*, *C*) of a subcomplex *S* of a complex *C relative to C* is the subcomplex of *C* containing all cells *c* of *C* whose closure Cl(*c*, *C*) contains both cells of *S* as well as cells of the complement *C*–*S*.

Definition BD: The (combinatorial) boundary ∂S of an *n*-dimensional subcomplex *S* of a complex *C* is the union of the closures of all $(n-1)$-cells of *C* each of which bounds exactly one *n*-cell of *S*.

Definition HN: An *n*-dimensional subcomplex *S* of a complex *C* is called *homogeneously n-dimensional* iff each cell of *S* bounds at least one *n*-cell of *S*.

Definition RG: An *n*-dimensional subcomplex *S* of a complex *C* is called a *region* of *C* iff it is homogeneously *n*-dimensional and $C-S-\partial C$ is also homogeneously *n*-dimensional.

Definition TL: A connected one-dimensional complex whose each cell, except two of them, is incident to exactly two other cells, is called a *topological line*.

It is easily seen that it is possible to assign integer numbers to the cells of a topological line in such a way that a cell incident to the cell having the number *k* has the number $k-1$ or $k+1$. These numbers are called the *topological coordinates* of the cells [8].

Definition CR: A Cartesian (direct) product C^n of n topological lines is called an n-dimensional *Cartesian* complex [8].
The set of cells of C^n is the Cartesian product of n sets of cells of the topological lines which are the *coordinate axes* of the n-dimensional space C^n. They are denoted by A_i, i=1,2,...,n. A cell of C^n is an n-tuple (a_1, a_2,..., a_n) of cells a_i of the corresponding axes: $a_i \in A_i$. The bounding relation of C^n is defined as follows: the n-tuple (a_1, a_2,..., a_n) is bounding another distinct n-tuple (b_1, b_2,..., b_n) iff for all i=1,2,...n the cell a_i is incident to b_i in A_i and $dim(a_i) \leq dim(b_i)$ in A_i.

The dimension of a product cell is defined as the sum of dimensions of the factor cells in their one-dimensional spaces. Topological coordinates of a product cell are defined by the vector whose components are the coordinates of the factor cells in their axes.

Fig. A.1a shows four cells in a two-dimensional Cartesian complex: P is a 0-cell (point), C_1 and C_2 are 1-cells (a horizontal and a vertical crack), F is a 2-cell (pixel).

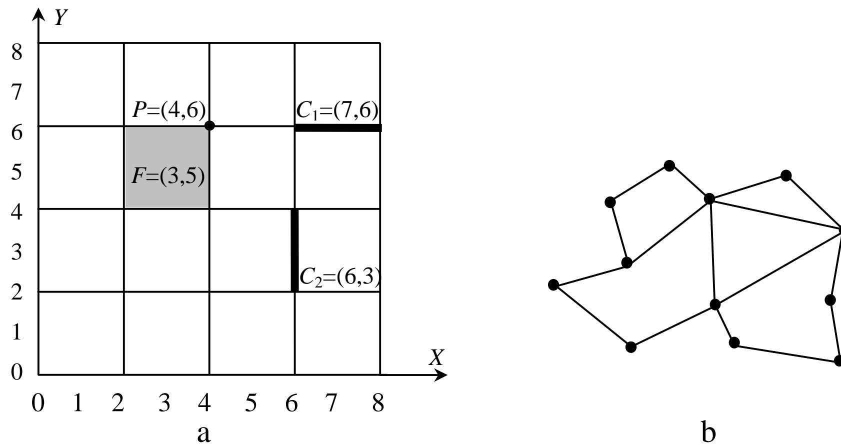


Fig. A.1. Example of a two-dimensional Cartesian (a) and non-Cartesian (b) complexes

If we assign even numbers to the 0-cells and odd ones to the 1-cells of the axes then the dimension of a cell in a Cartesian complex is equal to the number of its odd coordinates.