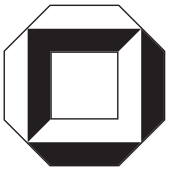


UNIVERSITÄT KARLSRUHE

Labeling of *n*-dimensional images with choosable adjacency of the pixels

K. Sandfort
J. Ohser

Preprint Nr. 06/07

Institut für Wissenschaftliches Rechnen
und Mathematische Modellbildung

76128 Karlsruhe

Anschriften der Verfasser:

Kai Sandfort
Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung
Universität Karlsruhe
D-76128 Karlsruhe

Prof. Dr. Joachim Ohser
Fachbereich Mathematik und Naturwissenschaften
Hochschule Darmstadt
D-46295 Darmstadt

Labeling of n-dimensional images with choosable adjacency of the pixels

Kai Sandfort[1] and Joachim Ohser[2]

August 19, 2006

Abstract

The labeling of discretized image data is one of the most essential operations in digital image processing. The notions of an adjacency system of pixels and the complementarity of two such systems are crucial to guarantee consistency of any labeling routine. In to date's publications this complementarity usually is defined using discrete versions of the Jordan-Veblen curve theorem and the Jordan-Brouwer surface theorem for two- and three-dimensional images, respectively. We here do a thorough analysis following an alternative concept which relies on a consistency relation for the Euler number. Exploiting this feature, which can be defined for any dimension in a uniform manner, we think that this is a more natural and powerful approach. We give the necessary definitions for the n-dimensional case and present identification and convergence results for complementary adjacency systems. An extensive discussion of a pseudo code framework for a general labeling algorithm finishes our paper, which puts the basis for a uniform treatment of images of arbitrary dimensionality.

Keywords: Labeling, run length encoding, adjacency system, complementarity, connectivity

1 Introduction

Labeling of connected components ('objects') is one of the most important tools of image processing. It is the basis for the generation of object features as well as of some kind of filtering, i. e. removing of noisy objects or holes in objects, where the criteria for an object or hole to be removed can be chosen extremely flexible based on the object features. The task of labeling (object filling, region detection) is to assign labels (in most cases unsigned integers) to the pixels in such a way that all pixels belonging to a connected component of the image are assigned the same label, and pixels belonging to different components have different labels.

Due to its importance in image processing, there is much literature about labeling and techniques to control (and improve) the processing and memory demands, which can be tremendous for large images frequently arising in practice. There still seems to be a lack of methods providing a satisfying combination of usability, flexibility, and efficiency. The prototype of labeling algorithms is the simple and well-known Rosenfeld-Pfaltz method [17, 8]. Here, the image is scanned until a pixel x_k is found that has not yet been labeled. If there is no neighboring pixel labeled with respect to the chosen adjacency system, a new label is chosen for x_k. Otherwise, if there is a neighboring pixel x_j with the label ℓ_j, the label ℓ_j is assigned also to x_k. In the case of more than one neighboring pixels having different labels, these labels are merged, which is noted in a table of pairs of equivalent labels. Finally, pixels belonging to the same equivalence class of labels form a connected component. The table of pairs can become very large, which may lead to problems in finding the equivalence classes; the complexity of a corresponding algorithm is $\mathcal{O}(m \log m)$ where m is the number of table entries. Thus, there are various versions of the Rosenfeld-Pfaltz method using techniques to keep m as small as possible.

Recent, more developed labeling algorithms usually comprise a preprocessing step as an essential part. It gives either a decomposition (see e.g. [1]) or a more compact representation (our method) of the input image and by that allows an efficient data access or just needs less memory and can be easily decompressed. Most techniques, including the Rosenfeld-Pfaltz method and our one, are

[1]University of Karlsruhe, IWRMM, Engesserstraße 6, D-76131 Karlsruhe
[2]University of Applied Sciences, Darmstadt, Schöfferstraße 3, D-46295 Darmstadt

2-pass-techniques, that means they run twice through the image, where in the first pass preliminary labels are assigned and label correspondences (pairs of equivalent labels) are collected, and in the second pass these correspondences are resolved into equivalence classes and the final labels are set. The resolving step is a critical issue and several methods have been proposed for this, some of which are described in [6]. A very efficient algorithm, which we adopt later on in our code, is explained in [21]. Instead of building the equivalence classes from all correspondences in the second run, one can do this during the first run by capturing every (preliminary) label in a new class and merging classes each time a correspondence occurs. Here, each label is internally mapped onto its current equivalence class identifier, so redundant equivalences can be avoided. This idea has been exposed in [6]. When applying this in classical algorithms where every pixel is tested for correspondences, it can achieve a big performance improvement. Since the situation is quite different in our algorithm, we have not (yet) incorporated it. An alternative way to reduce the number of label equivalences is discussed in [13], it relies on a partitioning of the image and a divide-and-conquer technique. In the method proposed in [3], a recursion step identifies a complete set of connected pixels (an object) and avoids the explicit construction of equivalence classes. A completely different labeling concept is followed by a single-pass-technique as described e.g. in [5], where contour tracing is used. Although this method type eliminates the additional access to pixels for relabeling, the tracing gets very complex (and inappropriate) for images of dimensionality ≥ 2. We finally remark that the introduction in [21] gives a short overview of different labeling algorithms, and Chapter 6 in [15] formalizes labeling operations on the basis of image algebra.

Our method includes a run length encoding of the input image before the actual labeling. This preprocessing is a central feature in our approach, it compresses the data and accelerates the access to it. As we explain in Section 5, it also is the evident reduction to the information which is needed for the labeling. The idea behind the run length encoding represents a natural property of the adjacency system or, equivalently, of an associated neighborhood graph, upon which the whole procedure bases (see the inclusion (3) and the first paragraph in Section 5).

The paper is organized as follows. Section 2 gives a short introduction to lattices (point lattices, grids). In order to give a general definition of pixel neighborhood, we follow the approaches of [11, 12, 19] and give a clear definition of adjacency on n-dimensional lattices and pairs of complementary adjacency systems, see Section 3. Section 4 completes the discussion of the theory necessary to legitimate and explain our labeling approach. In Section 5 we introduce the mathematical structures needed in our algorithm and clarify their meaning within the theoretical framework. Afterwards we state the run length encoding and the main labeling routine as easy pseudo codes, where the latter bases on the whole exposition before. We close this section by a short explanation of helper routines, which encapsulate some of the basic functionality. Finally, in Section 6 we discuss the capability and performance of our approach in comparison with other algorithms, present further applications of the run length encoding in image processing, and propose some important optimizations.

2 Homogeneous lattices

An n-dimensional homogeneous lattice is a subset $\mathbb{L}^n$ of the n-dimensional Euclidean space $\mathbb{R}^n$ with

$$\mathbb{L}^n = \{x \in \mathbb{R}^n : x = \sum_{i=1}^{n} \lambda_i u_i,\ \lambda_i \in \mathbb{Z}\} = U\mathbb{Z}^n \tag{1}$$

where $u_1, \ldots, u_n \in \mathbb{R}^n$ form a basis of $\mathbb{R}^n$, $U = (u_1, \ldots, u_n)$ is the matrix of column vectors, and $\mathbb{Z}$ ist the set of integers, see Figure 1. The closed unit cell of $\mathbb{L}^n$ *with respect to the basis* $\{u_1, \ldots, u_n\}$ is the Minkowski sum $C = [0, u_1] \oplus \ldots \oplus [0, u_n]$ of the segments $[0, u_i] = \{pu_i : 0 \leq p \leq 1\}$ between the origin 0 and the lattice points u_i. Its volume is $\operatorname{vol} C = |\det U| > 0$, and the value of $|\det U|$ does not depend on the choice of the basis. We denote by $\mathcal{F}^0$ the set of vertices of a polyhedron, in particular $\mathcal{F}^0(C) = U \cdot \{0, 1\}^n$. The set $\{C + x : x \in \mathbb{L}^n\}$ of all lattice cells covers $\mathbb{R}^n$, i.e. $\mathbb{R}^n = \bigcup_{x \in \mathbb{L}^n} (C + x)$. For more details and facts concerning lattices and their bases see e. g. [4] or [7].

Notice that for a given homogeneous lattice, the basis is not uniquely determined, and some of the notions, such as lattice spacing, unit cell, section lattice, that we use in the following, can depend on the choice of the basis.

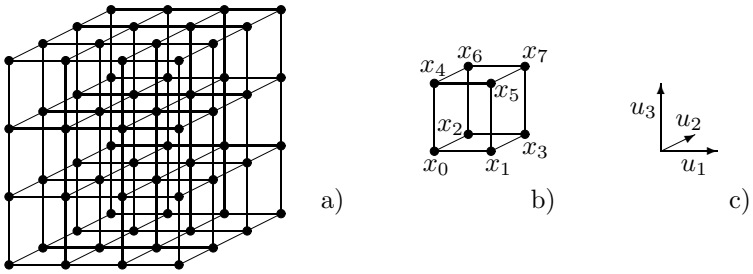


Figure 1: (a) a part of an orthorhombic primitive lattice $\mathbb{L}^3$, (b) a unit cell with the vertices $x_0, \ldots, x_7$, (c) the corresponding basis u_1, u_2, u_3.

3 Adjacency and Euler number

In the literature, the adjacency of lattice points (or pixels) is usually characterized by a neighborhood graph Γ, and the complementarity of adjacencies is defined via the Jordan-Brouwer surface theorem, see e. g. [9]. Here we use an alternative concept of adjacency systems based on the Euler number of a discretization, thoroughly introduced in [10, 11, 12].

3.1 Discretization with respect to an adjacency system

Let $\mathbb{L}^n$ be a lattice with the basis $\{u_1, \ldots, u_n\}$ and the unit cell C. The vertices of C are indexed, and we write $x_j = \sum_{i=1}^n \lambda_i u_i$, $\lambda_i \in \{0, 1\}$, with the index $j = \sum_{i=1}^n 2^{i-1}\lambda_i$. Clearly, the unit cell C has 2^n vertices, $x_i \in \mathcal{F}^0(C)$, $i = 0, \ldots, 2^n - 1$. In a similar way we introduce the index of a subset $\xi \subseteq \mathcal{F}^0(C)$. Let 1 denote the indicator function of a set, i. e. $1(x \in \xi) = 1$ if $x \in \xi$ and $1(x \in \xi) = 0$ otherwise. The index ℓ is assigned, and we write ξ_ℓ if

$$\ell = \sum_{j=0}^{2^n-1} 2^j \cdot 1(x_j \in \xi), \tag{2}$$

i. e. $\ell \in \{0, \ldots, \nu\}$ with $\nu = 2^{2^n} - 1$. Notice that $\xi_0 = \emptyset$, $\xi_\nu = \mathcal{F}^0(C)$, and $\xi_{\nu-\ell} = \xi_\nu \setminus \xi_\ell$. The ξ_ℓ can be considered as a local pixel configuration of the foreground of a binary image. Finally, we introduce the convex hulls $F_\ell = \operatorname{conv} \xi_\ell$ forming convex polytopes with $F_\ell \subseteq C$ and $\mathcal{F}^0(F_\ell) \subseteq \mathcal{F}^0(C)$, $\ell = 1, \ldots, \nu$. Let $\mathcal{F}^j(F)$ denote the set of all j-dimensional faces of a convex polytope F. For a set $\mathbb{F}$ of convex polytopes we set $\mathcal{F}^j(\mathbb{F}) = \cup\{\mathcal{F}^j(F) : F \in \mathbb{F}\}$.

Now we are able to equip the lattice $\mathbb{L}^n$ with a (homogeneous) adjacency system defining the neighborhood of lattice points.

Definition 1 *Let $\mathbb{F}_0 \subseteq \{F_0, \ldots, F_\nu\}$ be a set of convex polytopes $F_\ell = \operatorname{conv} \xi_\ell$ with the properties*

(i) $\emptyset \in \mathbb{F}_0$, $C \in \mathbb{F}_0$,

(ii) *if $F \in \mathbb{F}_0$, then $\mathcal{F}^i(F) \subset \mathbb{F}$ for $i = 0, \ldots, \dim F$,*

(iii) *if $F_i, F_j \in \mathbb{F}_0$ and $F_i \cup F_j$ is convex, then $F_i \cup F_j \in \mathbb{F}_0$.*

Then $\mathbb{F}_0$ is a local adjacency system and

$$\mathbb{F} = \bigcup_{x \in \mathbb{L}^n} \mathbb{F}_0 + x$$

is called an adjacency system on the lattice $\mathbb{L}^n$.

The pair $\Gamma = (\mathcal{F}^0(\mathbb{F}), \mathcal{F}^1(\mathbb{F}))$ is said to be the neighborhood graph of $\mathbb{F}$; it consists of the set $\mathcal{F}^0(\mathbb{F})$ of nodes and the set $\mathcal{F}^1(\mathbb{F})$ of edges. The order of the nodes is called the connectivity of $\mathbb{L}^n$.

In the simplest case, where the adjacency system is generated from the unit cell C, the order of the nodes is $2n$, and we write $\mathbb{F}_{2n} = \cup_{x \in \mathbb{L}^n} \cup_{j=0}^n \mathcal{F}^j(C + x)$. The maximum adjacency system

consisting of the convex hulls of all point configurations provides a κ-adjacency with $\kappa = 3^n - 1$, $\mathbb{F}_\kappa = \cup_{x\in\mathbb{L}^n}\{F_0 + x, \ldots, F_\nu + x\}$. Notice that for all adjacency systems $\mathbb{F}$ on $\mathbb{L}^n$ the inclusion

$$\mathbb{F}_{2n} \subseteq \mathbb{F} \subseteq \mathbb{F}_\kappa \tag{3}$$

holds. Now we recall the adjacency systems on $\mathbb{L}^3$ considered in detail in [11, 12].

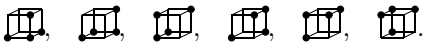
Examples ($n = 3$)

6-adjacency. The 6-adjacency is used as a standard in image processing. It is generated from the unit cell C, $\mathbb{F}_6 = \cup_{x\in\mathbb{L}^3} \cup_{j=0}^{3} \mathcal{F}^j(C + x)$.

14.1-adjacency. This adjacency system is generated from the tessellation of C into the 6 tetrahedra F_{139}, F_{141}, F_{163}, F_{177}, F_{197}, and F_{209}, which are the convex hulls of the configurations

i. e. $\mathbb{F}_0$ consists of all j-faces of the tetrahedra, $j = 0, \ldots, 3$, and their convex unions. The edges of the corresponding neighborhood graph Γ are the edges of C, the face diagonals of C containing the origin 0, the space diagonal of C containing 0, and all their lattice translations. The order of the nodes of Γ is 14.

14.2-adjacency. The 14.2-adjacency system is generated from the tetrahedra F_{43}, F_{141}, F_{147}, F_{169}, F_{177}, and F_{212}, which are the convex hulls of



The corresponding neighborhood graph Γ differs from that one for 14.1 in the choice of one face diagonal of C such that it does not contain 0.

26-adjacency. This system is given by $\mathbb{F}_\kappa = \mathbb{F}_{26} = \cup_{x\in\mathbb{L}^3}\{F_0 + x, \ldots, F_{255} + x\}$.

It should be noted that for $n > 2$ there can be two or more adjacency systems having the same neighborhood graph. In other words, an adjacency system $\mathbb{F}$ is not uniquely determined by Γ, see [19].

Definition 2 *The discretization $X \sqcap \mathbb{F}$ of a compact subset $X \subset \mathbb{R}^n$ with respect to a given adjacency system $\mathbb{F}$ is defined as the union of all j-faces of the elements of $\mathbb{F}$ for which all the vertices hit X, i. e.*

$$X \sqcap \mathbb{F} = \cup\{F \in \mathbb{F} : \mathcal{F}^0(F) \subseteq X\}. \tag{4}$$

This means that a 'brick' $F \in \mathbb{F}$ is a subset of the discretization of X if and only if all vertices of F belong to X, see [19].

It is important to realize that in particular for higher-dimensional images the connectivity of the pixels and, hence, the labeling can heavily depend on the choice of the adjacency system. The number of neighbors of a pixel in an n-dimensional image can range from $2n$ to $3^n - 1$. For the two- and three-dimensional case, the extremal choices are the well-known 4- and 8-connectivity, while for the three-dimensional case these are the 6- and 26-connectivity. As a consequence of the wide range of the number of neighbors, the neighborhood of pixels should be chosen very carefully in dependence of the dimensionality of the image, the lateral resolution, the image data, and the aims of processing and analysis.

3.2 Euler number and complementarity of adjacency systems

Since the set $X \sqcap \mathbb{F}$ forms a (not necessarily convex) polyhedron, the number $\sharp\mathcal{F}^j(X \sqcap \mathbb{F})$ of elements of $\mathcal{F}^j(X \sqcap \mathbb{F})$ is finite and, therefore, the Euler number $\chi(X \sqcap \mathbb{F})$ can be computed via the Euler-Poincaré formula,

$$\chi(X \sqcap \mathbb{F}) = \sum_{j=0}^{n} (-1)^j \, \sharp\mathcal{F}^j(X \sqcap \mathbb{F}). \tag{5}$$

A local version of (5) is given in [19].

It is well-known from image processing that if one chooses an adjacency system $\mathbb{F}$ on the discretization of X, then there is implicitly chosen a system $\mathbb{F}_c$ on the discretization of the complementary set X^c. In other words, if the 'foreground' $X \cap \mathbb{L}^n$ is connected with respect to $\mathbb{F}$, then the 'background' $X^c \cap \mathbb{L}^n$ must be connected with respect to $\mathbb{F}_c$. For $n > 2$ it is not sufficient to consider connectivity, and further criteria have to be regarded. In the following we introduce 'complementarity' by means of the Euler number of the discretization $X \sqcap \mathbb{F}$.

Let $\overline{X}$ denote the topological closure of X. In the continuous case the consistency relation $\chi(X) = (-1)^{n+1}\chi\left(\overline{X^c}\right)$ is fulfilled for all compact, polyconvex, and topologically regular sets $X \subset \mathbb{R}^n$, see [11, 14], and a similar relationship should hold in the discrete case.

Definition 3 *The pair $(\mathbb{F}, \mathbb{F}_c)$ is called a pair of complementary adjacency systems if*

$$\chi(X \sqcap \mathbb{F}) = (-1)^{n+1}\chi(X^c \sqcap \mathbb{F}_c) \tag{6}$$

holds for all compact $X \subset \mathbb{R}^n$. An adjacency system $\mathbb{F}$ is called self-complementary if $\chi(X \sqcap \mathbb{F}) = (-1)^{n+1}\chi(X^c \sqcap \mathbb{F})$ holds for all compact X.

A more simple criterion to check the complementarity of two adjacency systems is presented in [19]. However, for a given adjacency system $\mathbb{F}$ there does not necessarily exist an adjacency system $\mathbb{F}_c$ such that (6) holds. Furthermore, until now there is no constructive way to find the complementary system $\mathbb{F}_c$.

Examples ($n = 3$)

$(\mathbb{F}_6, \mathbb{F}_{26})$, $(\mathbb{F}_{14.1}, \mathbb{F}_{14.1})$ *and* $(\mathbb{F}_{14.1}, \mathbb{F}_{14.1})$. The 6-adjacency is complementary to the 26-adjacency. However, there are two self-complementary adjacency systems known, the 14.1-adjacency and the 14.2-adjacency, see [11, 12].

18-adjacency. Consider now a lattice $\mathbb{L}^3$ equipped with a neighborhood graph $\Gamma' = (\mathbb{L}^3, \mathcal{F}^1)$ where the system of edges $\mathcal{F}^1$ may consist of all edges and face diagonals of the cells of $\mathbb{L}^3$. The order of the nodes of Γ' is 18 and, hence, the adjacency is called the 18-adjacency (which is widely used in image processing). The 18-adjacency is 'Jordan-Brouwer-complementary' to the 6-adjacency generated solely from the edges of the lattice cells [9]. However, (6) does not hold for the pair $(\mathbb{F}_{18}, \mathbb{F}_6)$ and hence it is not complementary in the sense of Definition 3, see [19].

The last example shows that in higher dimensions ($n > 2$) the 'Jordan-Brouwer-complementarity' differs from the complementarity in Definition 3.

3.3 Multigrid convergence

Now we consider the relationship between the Euler number of a compact set $X \subset \mathbb{R}^n$ and the Euler number of its discretization. It can not be expected that $\chi(X) = \chi(X \sqcap \mathbb{F})$ for all compact sets X, but if X has a sufficiently smooth surface, the Euler number of $X \sqcap \mathbb{F}$ converges to the Euler number of X for increasing lateral resolution. Here 'smooth' is defined by morphological opening and morphological closure. The set X is called morphologically open with respect to a set $A \subset \mathbb{R}^n$ if X is invariant with respect to opening with A, $X \circ A = X$. Here the opening is defined by $X \circ A = (X \ominus \check{A}) \oplus A$, and $X \ominus A = (X^c \oplus A)^c$ is the Minkowski subtraction. Analogously, the set X is called morphologically closed with respect to A if $X \bullet A = X$ where $X \bullet A = (X \oplus \check{A}) \ominus A$ is the morphological closure with A.

Theorem 1 *Let $(\mathbb{F}, \mathbb{F}_c)$ be a pair of complementary adjacency systems on $\mathbb{L}^n$. If $X \subset R^n$ is morphologically closed with respect to all edges $F \in \mathcal{F}^1(\mathbb{F})$ and morphologically open with respect to all $F \in \mathcal{F}^1(\mathbb{F}_c)$, then*

$$\chi(X) = \chi(X \sqcap \mathbb{F}) \qquad \textit{and} \qquad \chi(X^c) = \chi(X^c \sqcap \mathbb{F}_c).$$

A proof is given in [11]. Notice that a set X fulfilling the last condition is polyconvex and, hence, its Euler number exists. However, this condition for X is very strong, it depends on $\mathbb{F}$ and, hence, it will not be fulfilled in most applications. Thus, we consider a more natural condition for X.

Let B_ε be a (small) ball of radius ε. A set X is said to be morphologically regular if there is an $\varepsilon > 0$ such that X is morphologically open as well as morphologically closed with respect to B_ε, $X \circ B_\varepsilon = X = X \bullet B_\varepsilon$. From Theorem 1 we obtain the following lemma.

Lemma 1 *Let $(\mathbb{F}, \mathbb{F}_c)$ be a pair of complementary adjacency systems on $\mathbb{L}^n$. Then $a\mathbb{F}$ is an adjacency system on $a\mathbb{L}^n$, $a > 0$, and it is*

$$\lim_{a \to 0} \chi(X \sqcap a\mathbb{F}) = \chi(X) \tag{7}$$

for all compact and morphologically regular sets X.

This means that the Euler number is convergent for morphologically regular sets (multigrid convergence). The proof of this lemma follows from the fact that if X is morphologically regular, there exists an $a > 0$ such that $\chi(X \bullet F) = \chi(X)$ for all $F \in a\mathbb{F}$ and $\chi(X \circ F) = \chi(X)$ for $F \in a\mathbb{F}_c$, and choose an $a > 0$ such that $F \subset B_\varepsilon$ for all $F \in a\mathbb{F} \cup a\mathbb{F}_c$.

4 Connectedness

In order to describe a labeling algorithm, it is necessary to introduce the notions of 'connectivity' and 'connected component'. These are provided by topology. Azriel Rosenfeld introduced a digital topology on $\mathbb{L}^2$, see [16]. He defined connectedness on lattices and stated a discrete Jordan-Veblen curve theorem. The definition of connectedness can simply be extended to n-dimensional lattices, see e. g. [9]. Here we apply the definition of an adjacency system from Section 3.1.

4.1 Continuous case

Firstly we consider the continuous case and introduce connectivity for the Euclidean space $\mathbb{R}^n$. The connected components of a bounded set $X \subset \mathbb{R}^n$ can be considered as the equivalence classes of $X \subseteq \mathbb{R}^n$ with respect to an appropriately chosen equivalence relation $\sim$ defined for point pairs in $\mathbb{R}^n$.

Definition 4 *A set $X \subset \mathbb{R}^n$ is said to be connected if for all subsets $X_1, X_2 \subseteq X$ with $X_1 \cup X_2 = X$ it follows that $\overline{X}_1 \cap X_1 \neq \emptyset$ or $X_1 \cap \overline{X}_2 \neq \emptyset$.*

This definition of connectivity is closely related to path-connectivity. A path in $\mathbb{R}^n$ is a continuous mapping $f : [0,1] \mapsto \mathbb{R}^n$. If $f(0) = x$ and $f(1) = y$, $x, y \in \mathbb{R}^n$, then f is called a path from x to y.

Definition 5 *A non-empty set X is called path-connected if for every $x, y \in X$ there exists a path f from x to y such that $f(\cdot) \subseteq X$.*

It is well-known that every path-connected set X is also connected. Furthermore, if X is open and connected, it is also path-connected. Obviously, a connected set X is not necessarily path-connected. For example, the curve of the function $\sin(1/x)$ is connected, but not path-connected. More precisely, the set $X = \{(x, \sin(1/x)) : x \in \mathbb{R}\backslash\{0\}\} \cup \{0\}$ is connected, but not path-connected, see [20].

We write $x \sim y$ for path-connected points $x, y \in \mathbb{R}^n$. It can be shown that the binary relation $\sim$ is an equivalence relation, i. e. $\sim$ is reflexive, symmetric, and transitive. The equivalence classes $X_1, \ldots, X_m$ of X under $\sim$ are called path components of X. For more details see e. g. [2, 18].

4.2 Discrete case

Connectedness in a discretization is closely related to adjacency of lattice points. Hence, we consider a homogeneous lattice $\mathbb{L}^n$ equipped with a pair of complementary adjacency systems $(\mathbb{F}, \mathbb{F}_c)$. Let x and y be lattice points, $x, y \in \mathbb{L}^n$. A discrete path from x to y with respect to the adjacency system $\mathbb{F}$ is a sequence of lattice points $(x_i)_{i=0}^m \subset \mathbb{L}^n$, $m \in \mathbb{N}$, with $x_0 = x$, $x_m = y$, and $[x_{i-1}, x_i] \in \mathbb{F}$, $i = 1, \ldots, m$.

A non-empty discrete set $Y \subseteq \mathbb{L}^n$ is called path-connected with respect to $\mathbb{F}$ if $\sharp Y = 1$ or if for all pairs $(x, y) \in Y^2$ with $x \neq y$ there exists a discrete path with respect to $\mathbb{F}$ from x to y. Connectedness with respect to $\mathbb{F}$ in Y is an equivalence relation.

Definition 6 *Let $\mathbb{L}^n$ be a homogeneous lattice equipped with an adjacency system $\mathbb{F}$, and let $Y \subseteq \mathbb{L}^n$ be a discrete set. The equivalence classes $Y_1, \ldots, Y_m \subseteq Y$, $m \geq 1$, defined through the connectedness with respect to $\mathbb{F}$ are called the connected components of Y.*

We will use the notation $Y_{\mathbb{F}} = \{Y_1, \ldots, Y_m\}$ for the set of equivalence classes of Y with respect to $\mathbb{F}$. As a consequence of Lemma 1 we obtain the following result.

Lemma 2 *Let $(\mathbb{F}, \mathbb{F}_c)$ be a pair of complementary adjacency systems on $\mathbb{L}^n$, and let X be a compact and morphologically regular subset of $\mathbb{R}^n$ with the set of equivalence classes $\{X_1, \ldots, X_m\}$ under $\sim$. Then there is a constant $b > 0$ such that*

$$(X \cap a\mathbb{L}^n)_{a\mathbb{F}} = \{X_1 \cap a\mathbb{L}^n, \ldots, X_m \cap a\mathbb{L}^n\} \tag{8}$$

for all a with $0 < a < b$.

In other words, for sufficiently high lateral lattice resolution the equivalence classes of $(X \cap a\mathbb{L}^n)$ are independent of the choice of the adjacency system. However, this holds only for sets X with sufficiently smooth surface. In general, the equivalence classes of $X \cap \mathbb{L}^n$ depend on $(\mathbb{F}, \mathbb{F}_c)$.

5 The Labeling Algorithm

In this and the following section, we will turn our attention to the implementation and discussion of a general, customizable labeling algorithm. This bases on the theoretical background developed in the previous sections and adopts its notation. Additional variables and data structures used in our realization are explained in the respective places. We present the routines as easy translatable C-style pseudo codes.

Before we give precise definitions of the data structures below, we will shortly describe the labeling procedure. To this end, we anticipate that an image $Y \subset \mathbb{L}^n$ is a finite discrete set of lattice points (the foreground pixels of a binary image).

Preceding the actual labeling, our algorithm does a run length encoding of Y to give a compact representation and accelerate the access to the elements of Y. As a 'run' we here consider a set of consecutive lattice points $x \in \mathbb{L}^n$ in a certain dimension of Y. The idea behind this approach is the fact that the elements of a run are naturally considered as part of the same connected component. This is expressed in the inclusion (3) and thus e. g. complies with all local adjacency systems given on page 4 for the case $n = 3$. All correspondences, i. e. identifications of elements of Y belonging to the same connected component, can be completely acquired by checking the starting (lattice) point and the endpoint of a run for such correspondences, according to the specific local adjacency system presumed.

To finally put it in formal terms, using the notation introduced above, a labeling is a mapping from $Y \subset \mathbb{L}^n$ to $\mathbb{N}$ where each lattice point $x \in Y_k$ is assigned the index k, $k = 1, \ldots, m$. Thus each labeling depends on $(\mathbb{F}, \mathbb{F}_c)$.

5.1 The model for the data structures

For the following discussion, we here formulate the model for the data structures necessary to handle image data arising in practice and in order to formalize the operations on it.

The model. Using the notations $U = (u_1, \ldots, u_n)$ and $U_j = (u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_n)$, we define the following structures:

the window: $W = [0, m_1 u_1] \oplus \ldots \oplus [0, m_n u_n], \qquad m_1, \ldots, m_n \in \mathbb{N},$

the image: $Y = \mathbb{L}^n \cap X \cap W \qquad$ where $\mathbb{L}^n = U\mathbb{Z}^n$,

a block: $B_{\tilde{W}} = \mathbb{L}^n \cap X \cap \tilde{W} = Y \cap \tilde{W} \quad$ where $\tilde{W} = [\tilde{m}_1^{\min} u_1, \tilde{m}_1^{\max} u_1] \oplus \ldots \oplus [\tilde{m}_n^{\min} u_n, \tilde{m}_n^{\max} u_n]$
with $\tilde{m}_j^{\min}, \tilde{m}_j^{\max} \in \mathbb{N}$ and $0 \leq \tilde{m}_j^{\min} \leq \tilde{m}_j^{\max} \leq m_j$ for all $j \in \{1, \ldots, n\}$, i.e. $\tilde{W} \subseteq W$,

projectors: $P_j(\mathbb{L}^n) = U_j\mathbb{Z}^{n-1}, \qquad i = 1, \ldots, n$
and $P_M(\mathbb{L}^n) = \bigcap_{k \in M} P_k(\mathbb{L}^n) \quad$ for $M \subseteq \{1, \ldots, n\}$,

a subimage: $T_{\hat{M},z} = \big(z + P_{\hat{M}}(\mathbb{L}^n)\big) \cap X \cap W \quad$ for $z \in \mathbb{L}^n$, $\hat{M} \subseteq \{1,\dots,n\}$,

a slice: $S_{\tilde{M},y} = (y + P_{\tilde{M}}(\mathbb{L}^n)) \cap X \cap W \quad$ for $y \in \mathbb{L}^n$, $\sharp\tilde{M} = n-2$, $\tilde{M} \subset \{1,\dots,n\}$,

a line: $L_{M,x} = (x + P_M(\mathbb{L}^n)) \cap X \cap W \quad$ for $x \in \mathbb{L}^n$, $\sharp M = n-1$, $M \subset \{1,\dots,n\}$.

Here, $\sharp$ denotes the number of elements in the specified set. We point out that our definition of the structure 'image' is a restriction to sets of a special form, however this is really common and facilitates the description of many operations on it. Furthermore, we remark that

$$\begin{aligned}
S_{\tilde{M},y} = T_{\hat{M},z} \quad &\Leftrightarrow \quad \tilde{M} = \hat{M}, \quad z - y \in P_{\tilde{M}}(\mathbb{L}^n), \\
L_{M,x} = T_{\hat{M},z} \quad &\Leftrightarrow \quad M = \hat{M}, \quad z - x \in P_M(\mathbb{L}^n), \\
L_{M,x} = S_{\tilde{M}_1,y_1} \cap S_{\tilde{M}_2,y_2} \quad &\Leftrightarrow \quad \sharp(\tilde{M}_1 \cap \tilde{M}_2) = n-3, M = \tilde{N} \backslash (\tilde{N} \backslash \tilde{M}_1 \cap \tilde{N} \backslash \tilde{M}_2) \\
&\qquad \text{where } \tilde{N} = \{1,\dots,n\},\ x - y_1 \in P_{\tilde{M}_1}(\mathbb{L}^n),\ x - y_2 \in P_{\tilde{M}_2}(\mathbb{L}^n).
\end{aligned}$$

The first two equivalences state that a slice and a line are special types of a subimage. The third equivalence means that a line can be represented as the intersection of two slices.

For the considerations below, the following aspects are very important:

- For a n-dimensional image Y there are $2^n n!$ possibilities to scan through the n lattice directions of Y which are constituted by the basis elements $u_1,\dots,u_n$. The factorial in this term originates from the scanning order and the factor 2^n from the orientations of the directions. The order is specified by the *ranks* of the directions, where the first scanning direction has rank 0, the second one rank 1, etc.
- Runs of lattice points are detected and coded in the direction with rank 0. Particularly for strongly anisotropic sets X, the processing speed of the labeling heavily depends on the choice of this direction since this determines the number of runs.

Basic structures. In the pseudo code for our algorithms for the run length encoding and the labeling we use the following identifiers for global variables and data structures. To support an easy understanding of the code, scalar quantities begin with a small letter, and vectors as well as data structures begin with a capital letter. All vectors are assumed to provide the methods `add()` to add an element, `clear()` to delete all elements, `erase()` to delete the element addressed by the pointer argument, `initialize()` to set all elements to the value of the argument, `resize()` to resize the vector to the length indicated by the argument, and `size()` to query the number of elements.

As basic variables and data structures we now introduce `Image`, `Image_Size`, `Directions`, `Increments`, `Neighborhood`, `selection_value`, `RLE_Run`, and `RLE_Line`.

By `Image` we denote the image data container for a specific image Y. Each entry of this container is called a *pixel* and references a lattice point x in Y. The pixel values (i. e. the preliminary labels) are assumed to be accessible by `Image`$[x_1][x_2]\dots[x_n]$ where $(x_1,\dots,x_n)$ is the coordinate vector for x. The dimensions of `Image` are stored in `Image_Size`.

The vector `Directions` provides the ranks of the lattice directions and indicators for the orientations which are regarded while scanning through the image. The absolute value of `Directions`$[i]$ is the rank of the direction with index i (made up by u_i), and the sign of `Directions`$[i]$ determines its scanning orientation ('+' for the orientation given by increasing coordinates, '−' for the reverse orientation).

In addition, `Increments` stores the step sizes for pacing the lattice directions where `Increments`$[i]$ usually equals $+1$ (if the sign of `Directions`$[i]$ is '+') or -1 (if the sign of `Directions`$[i]$ is '−'). With larger absolute values of the steps it is possible to scan through a coarse lattice than that implicitly given by `Image`.

`Neighborhood` is a two-dimensional array which should be thought of as a vector of (coordinate) vectors. It keeps the relative coordinates of all pixels which are considered as neighbors refering to the current pixel. Each coordinate of each neighbor hence usually is either $+1$ or -1. We point out that this data structure is essentially connected with the local adjacency system as defined in Subsection 3.1. `Neighborhood`$[i][j]$ is the relative coordinate in the direction with index j of the $i+1$-th neighbor (for the first neighbor it is $i = 0$, the order in which the neighbors are stored in `Neighborhood` does not matter). Note that in this way general neighborhoods can be specified,

among which are those associated with local adjacency systems. Nevertheless it has to be noted that due to the run length encoding the pixels which immediately precede and follow the current pixel in the direction with rank 0 (according to the corresponding step size in `Increments`) are treated as neighbors and so should be contained in `Neighborhood`. Since the direction with rank 0 is intended to be customizable, this should apply to every direction. However, this restriction is quite natural and expressed in (3).

The scalar variable `selection_value` stores a pixel value by means of which pixels are selected which are to be encoded as runs. If this value is set to -1, then all runs of pixels with arbitrary value are detected and saved, otherwise only runs of pixels with the value given by `selection_value` are processed. The latter option especially is important for the case that `Image` contains a binary image, and `selection_value` should be set to the value of foreground pixels then.

The structure `RLE_Run`, saving the data of a single run, consists of the three members `length`, `pos`, and `label`. The member `length` clearly gives the length of the run, i.e. the number of constituting pixels. `pos` determines the starting position of the run. Note that this depends on the sign of `Directions[Rank_to_Index[0]]`. The value of the pixels of the run is saved in `label`. This will be changed to the label of the object comprising this run in `WriteLabeledImage()` later on. We remark that this member might be discarded if `Image` represents a binary image with predefined values for foreground and background pixels and `selection_value` is set to either of these.

Finally, `RLE_Line` represents a run length encoded line and has the single member `Runs`, which is a vector of `RLE_Run`s. The number of elements of this vector depends on the contents of the image, `Rank_to_Index[0]`, and `selection_value`.

5.2 The Run Length Encoding Algorithm

Besides the basic variables and data structures from above the implementation of our run length encoding uses the following quantities. The letter 'C' in the notation indicates, that the respective variable stores a single coordinate in a certain dimension, 'V' similarly refers to a single pixel value, and the abbreviation "Cs" is used in vectors which store the n (absolute or relative) coordinates of a pixel.

`counter` – an auxiliary counter variable for assigning a unique preliminary label to each run of pixels with the specified value `selection_value`, if this is `<> -1`, or to every run of pixels, otherwise.

`total_num_lines` – the total number of lines.

`Rank_to_Index` [global] – stores the order of the directions in which to pass through the image; it maps the rank of a direction to its index.

`Index_to_Rank` [global] – the complementary vector to `Rank_to_Index`, mapping the index of a direction to its rank i.e. `Rank_to_Index[Index_to_Rank[i]] = i` and `Index_to_Rank[Rank_to_Index[r]] = r`.

`Orientations` – stores indicators for the orientations ($+1$ = forward, -1 = backward) to be regarded while passing through the image.

`Line_Position` – stores the relative position of the current line in the image, with respect to the specified order of directions and their orientations, i.e.
`Current_Pixel_Cs[j] = (Orientations[r] == 1)? Line_Position[r-1]:`
`i_size[j]-1-Line_Position[r-1]` for `r > 0` with `j = Rank_to_Index[r]`.

`RLE_Data` [global] – an n-dimensional array of `RLE_Line`s storing the run length encoded version of the image.

`stop_main_loop` [global] – indicates when to stop the main loop passing through the image and processing the data.

`first_pixel_0_C` – the absolute coordinate of the first pixel in the dimension with rank 0.

`last_pixel_0_C` – the absolute coordinate of the last pixel in the dimension with rank 0.

`Current_RLE_Line` – stores the data of all runs in the current line and hence represents the run length encoded version of the current line.

`Current_Pixel_Cs` – the absolute coordinates of the current pixel according to the original order of directions in the image, i. e. `current_pixel_V = image[Current_Pixel_Cs[0]]...[Current_Pixel_Cs[n-1]]`.

`current_pixel_V` – stores the value of the current pixel.

`previous_pixel_V` – stores the value of the pixel preceding the current pixel, to detect runs.

`Current_Run` – stores the data of the current run.

Variables which are marked as global are assumed to be known in all subroutines called after their definition. With this preparation we turn to the run length encoding algorithm `DoEncoding()`. The small auxiliary routines `Transform()` and `Update()` are considered at the end of the next subsection, together with some other helper routines called by the labeling code. The algorithm looks as follows:

```
void DoEncoding()

{
  boolean stop_main_loop;
  unsigned long i, total_num_lines, first_pixel_0_C, last_pixel_0_C, tmp_pixel, tmp_index;
  long previous_pixel_V, current_pixel_V, counter;
  vector<char>[n] Orientations;
  vector<unsigned long>[n] Index_to_Rank, Rank_to_Index, Current_Pixel_Cs;
  vector<unsigned long>[n-1] Line_Position;
  multi_dim_array<RLE_Line>[] RLE_Data;
  RLE_Run Current_Run;
  RLE_Line Current_RLE_Line;

  // initialize the variables
  counter = 0;
  total_num_lines = 1;
  Rank_to_Index[0] = abs(Directions[0]);
  Index_to_Rank[Rank_to_Index[0]] = 0;
  Orientations[0] = sign(Directions[0]);
  Line_Position.initialize(0);

  FOR i FROM 1 TO n-1 DO
  {
    Rank_to_Index[i] = abs(Directions[i]);
    // The equality Rank_to_Index[Index_to_Rank[j]] = j, j = 1,...,n-1,
    // (see the explanation of the variable Index_to_Rank above)
    // is implied by the following command.
    Index_to_Rank[Rank_to_Index[i]] = i;
    Orientations[i] = sign(Directions[i]);
    total_num_lines *= Image_Size[Rank_to_Index[i]];
  }

  // The next command allocates memory for the array RLE_Data,
  // the size of the i-th dimension (i = 1,...,n-1) of RLE_Data is the size of the image
  // in the direction with rank n-i.
  // Runs in the direction with rank 0 are stored in the corresponding element of RLE_Data
  // of type RLE_Line.
  RLE_Data = new RLE_Line[Image_Size[Rank_to_Index[n-1]]][Image_Size[Rank_to_Index[n-2]]]... >
             > [Image_Size[Rank_to_Index[1]]];

  stop_main_loop = false;
  first_pixel_0_C = 0;
  last_pixel_0_C = Image_Size[Rank_to_Index[0]] - 1;

  // In the case that the direction with rank 0 should be passed backwards,
  // the first and the last pixel coordinate have to be swapped.
  IF Increments[Rank_to_Index[0]] < 0 THEN
  {
    tmp_pixel = first_pixel_0_C;
    first_pixel_0_C = last_pixel_0_C;
    last_pixel_0_C = tmp_pixel;
```

```
  }

  // main loop for passing through the image
  WHILE NOT stop_main_loop DO
  {
    Current_RLE_Line.Runs.clear();

    previous_pixel_V = -1;
    // The following command computes the coordinates in the directions with rank > 0
    // from the new line position.
    // These coordinates are the same for all pixels in the new line to be encoded.
    Transform(Current_Pixel_Cs, Line_Position);

    // loop through the current line with respect to the given step size
    FOR i FROM first_pixel_0_C TO last_pixel_0_C STEP Increments[Rank_to_Index[0]] DO
    {
      // set the missing coordinate of the current pixel in the direction with rank 0
      Current_Pixel_Cs[Rank_to_Index[0]] = i;
      // query the value of the current pixel
      current_pixel_V = Image[Current_Pixel_Cs[0]][Current_Pixel_Cs[1]]...[Current_Pixel_Cs[n-1]];

      // update the length of the current run, if the value of the current pixel is
      // the same as the value of the previous one, or create a new run, otherwise
      IF current_pixel_V == previous_pixel_V THEN
        Current_Run.length += 1;
      ELSE
      {
        // store the current run in Current_RLE_Line if it has an admissible preliminary label
        // (unequal to selection_value, if this is <> -1)
        IF previous_pixel_V <> -1 && (selection_value == -1 ||
        (selection_value <> -1 && previous_pixel_V <> selection_value)) THEN
          Current_RLE_Line.Runs.add(Current_Run);

        Current_Run.length = 1;
        Current_Run.pos = i;
        // assign the value of the starting pixel as preliminary label for the current run
        Current_Run.label = current_pixel_V;
      }

      previous_pixel_V = current_pixel_V;
    }

    // store the last run in the current line in Current_RLE_Line
    // if it has an admissible preliminary label
    IF selection_value == -1 ||
      (selection_value <> -1 && previous_pixel_V <> selection_value) THEN
      Current_RLE_Line.Runs.add(Current_Run);

    // assign a new preliminary label to every run in Current_RLE_Line
    // such that every run in the image gets a unique preliminary label
    FOR i FROM 1 TO Current_RLE_Line.Runs.size() DO
    {
      counter++;
      Current_RLE_Line.Runs[i-1].label = counter;
    }

    // store the run length encoded line, represented by Current_RLE_Line,
    // in the array RLE_Data
    RLE_Data[Line_Position[n-2]][Line_Position[n-3]]...[Line_Position[0]] = Current_RLE_Line;

    // update the line position
    // (stop_main_loop is modified in Update() if necessary)
    Update(Line_Position);
  }
}
```

5.3 Labeling with choosable adjacency

Now we consider the actual labeling of the image data which is based on a preceding application of the run length encoding We have already mentioned above that the labeling can be correctly done by solely testing the value of the starting point and the endpoint of runs for correspondences with

other pixel values. The proposed way to access the relevant image data accelerates the labeling process, especially by substantially reducing the number of value correspondences to be processed during the computation of the value-to-label-map (which is called `Label_Map` below). Further uses of this preprocessing step in multidimensional image processing will be discussed later in a separate section. In the pseudo code for the labeling we use the following additional variables, keeping the introduced notation style:

`current_run_V` – the value of the current run.

`current_neighbor_valid` – indicates whether a neighboring point is within the bounds of the image or not.

`current_neighbor_C` – successively stores each coordinate of the current neighboring point.

`First_Run_Pixel_Cs` – the starting point of the current run.

`current_neighbor_V` – the value of the currently tested neighboring point.

`Current_Neighbor_Cs` – current neighboring point.

`V_Pair` – stores a value correspondence, i. e. two values which are considered equivalent and will be mapped to the same label.

`Correspondences` – stores all value correspondences.

`Last_Run_Pixel_Cs` – the end point of the current run.

`Label_Map` [global] – the indices of the entries of this vector represent the values (preliminary labels) of runs, and the values of its entries are the final labels of the associated objects.

All variables not listed above have the same meaning as they have in `DoEncoding()`. With this we now present the labeling algorithm.

```
void DoLabeling()

{
  boolean stop_main_loop, current_neighbor_V;
  unsigned long i, j, k;
  long current_run_V, current_neighbor_V, current_neighbor_C;
  RLE_Run Current_Run;
  RLE_Line Current_RLE_Line;
  LabelCorrespondence V_Pair;
  vector<unsigned long>[n] First_Run_Pixel_Cs, Last_Run_Pixel_Cs, Current_Neighbor_Cs;
  vector<unsigned long>[n-1] Line_Position;
  vector<long> Label_Map;
  vector<LabelCorrespondence> Correspondences;

  // initialize the variables
  Line_Position.initialize(0);
  stop_main_loop = false;

  // main loop for passing through the image
  WHILE NOT stop_main_loop DO
  {
    // read a new run length encoded line from the array RLE_Data
    Current_RLE_Line = RLE_Data[Line_Position[n-2]][Line_Position[n-3]]...[Line_Position[0]];
    // The following command computes the coordinates in the directions with rank > 0
    // from the new line position.
    // These coordinates are the same for all starting and end points of runs in the new line
    // to be decoded.
    Transform(First_Run_Pixel_Cs, Line_Position);
    // set the end point (of any run) to the starting point except the coordinate in the
    // direction with rank 0
    FOR i FROM 1 TO n DO
      Last_Run_Pixel_Cs[i-1] = First_Run_Pixel_Cs[i-1];

    // loop through the current line to be decoded
    FOR i FROM 1 TO Current_RLE_Line.Runs.size() DO
    {
```

```
  Current_Run = Current_RLE_Line.Runs[i-1];
  // get the preliminary (unique) label from the current run
  current_run_V = Current_Run.label;
  // set the missing coordinate of the starting point of the current run in the direction
  // with rank 0
  First_Run_Pixel_Cs[Rank_to_Index[0]] = Current_Run.pos;

  // test the neighborhood of the starting point of the current run for label correspondences
  // with respect to the given neighborhood
  FOR j FROM 1 TO Neighborhood.num_neighbors DO
  {
    current_neighbor_valid = true;

    FOR k FROM 0 TO n-1 WHILE current_neighbor_valid DO
    {
      current_neighbor_C = First_Run_Pixel_Cs[k] + (Neighborhood.Neighbors[j-1])[k];
      // check whether the neighboring point exists
      // or the starting point lies at some boundary
      IF current_neighbor_C >= 0 AND current_neighbor_C < Image_Size[k] THEN
        Current_Neighbor_Cs[k] = current_neighbor_C;
      ELSE
        current_neighbor_valid = false;
    }

    // query the label of the current neighbor (if existing)
    IF current_neighbor_valid THEN
    {
      current_neighbor_V = QueryLabel(Current_Neighbor_Cs);
      IF current_neighbor_V <> -1 THEN
      {
        // store the label correspondence between the starting point and the current
        // neighboring point
        V_Pair.value1 = current_run_V;
        V_Pair.value2 = current_neighbor_V;
        Correspondences.add(V_Pair);
      }
    }
  }

  // repeat the above testing loop for the end point of the current run
  // if the run has a length > 1 (otherwise the starting and the end point coincide)
  IF Current_Run.length <> 1 THEN
  {
    Last_Run_Pixel_Cs[Rank_to_Index[0]] = Current_Run.pos + Current_Run.length - 1;

    FOR j FROM 1 TO Neighborhood.num_neighbors DO
    {
      current_neighbor_valid = true;

      FOR k FROM 0 TO n-1 WHILE current_neighbor_valid DO
      {
        current_neighbor_C = Last_Run_Pixel_Cs[k] + (Neighborhood.Neighbors[j-1])[k];
        IF current_neighbor_C >= 0 AND current_neighbor_C < Image_Size[k] THEN
          Current_Neighbor_Cs[k] = current_neighbor_C;
        ELSE
          current_neighbor_valid = false;
      }

      IF current_neighbor_valid THEN
      {
        current_neighbor_V = QueryLabel(Current_Neighbor_Cs);
        IF current_neighbor_V <> -1 THEN
        {
          V_Pair.value1 = current_run_V;
          V_Pair.value2 = current_neighbor_V;
          Correspondences.add(V_Pair);
        }
      }
    }
  }
}

// update the line position
```

```
    // (stop_main_loop is modified in Update() if necessary)
    Update(Line_Position);
  }

  // call the helper routines BuildLabelVector() and ResolveLabelVector()
  // to obtain the final value-to-label-map, represented by Label_Map, where
  // value means the preliminary (unique) label of a run and
  // label means the final label of the object comprising this run
  Label_Map = BuildLabelVector(Correspondences);
  ResolveLabelVector(Label_Map);
}
```

After the vector `Label_Map` has been generated by `DoLabeling()`, we are ready to finally write the labeled image by means of `Label_Map` and `RLE_Data`. This job is done by

```
void WriteLabeledImage()

{
  boolean stop_main_loop;
  unsigned long i, j;
  vector<unsigned long>[n] Current_Pixel_Cs;
  vector<unsigned long>[n-1] Line_Position;
  RLE_Run Current_Run;
  RLE_Line Current_RLE_Line;

  Line_Position.initialize(0);

  WHILE NOT stop_main_loop DO
  {
    Current_RLE_Line = RLE_Data[Line_Position[n-2]][Line_Position[n-3]]...[Line_Position[0]];
    Transform(Current_Pixel_Cs, Line_Position);

    FOR i FROM 1 TO Current_RLE_Line.Runs.size() DO
    {
      Current_Run = Current_RLE_Line.Runs[i-1];
      Current_Pixel_Cs[Rank_to_Index[0]] = Current_Run.pos;

      Current_Run.label = Label_Map[Current_Run.label];

      FOR j FROM 1 TO Current_Run.length DO
      {
        Image[Current_Pixel_Cs[0]][Current_Pixel_Cs[1]]...[Current_Pixel_Cs[n-1]] = Current_Run.label;
        Current_Pixel_Cs[Rank_to_Index[0]] += Increments[Rank_to_Index[0]];
      }
    }

    Update(Line_Position);
  }
}
```

Hence, the calling sequence for our total labeling procedure is `DoEncoding()` – `DoLabeling()` – `WriteLabeledImage()`. Having described these algorithms, we also shortly explain the auxiliary routines `BuildLabelVector()`, `ResolveLabelVector()`, `Update()`, `Transform()`, and `QueryLabel()`.

The following routine builds an incomplete label map by means of the correspondences, the collection of equivalent preliminary labels. To complete the map such that every possible label is mapped to its smallest equivalent label, `ResolveLabelVector()` has to be called afterwards. A similar procedure for resolving correspondences is described in [1].

```
vector<long> BuildLabelVector(vector<CLabelCorrespondence> Corresp)

{
  vector<long> Lab_Map;
  unsigned long i;
  long first_label, second_label, label, max_label;

  max_label = 0;

  FOR i FROM 0 TO Corresp.size()-1 DO
  {
    label = Max(Corresp[i].value1, Corresp[i].value2);
    IF label > max_label THEN
```

```
      max_label = label;
  }

  Lab_Map.resize(max_label+1);
  Lab_Map.initialize(0);

  FOR i FROM 0 TO Corresp.size()-1 DO
  {
    first_label = Corresp[i].value1;
    second_label = Corresp[i].value2;

    WHILE Lab_Map[first_label] <> 0 DO
      first_label = Lab_Map[first_label];

    WHILE Lab_Map[second_label] <> 0 DO
      second_label = Lab_Map[second_label];

    IF first_label < second_label THEN
      Lab_Map[second_label] = first_label;
    ELSE IF first_label > second_label THEN
      Lab_Map[first_label] = second_label;
  }

  return(Lab_Map);
}
```

The next routine 'cleans up' and completes the label map as it is output by `BuildLabelVector()`. As the final step, each label for which there is no smaller equivalent label is assigned the smallest value, which is not identified with another (non-equivalent) label.

```
void ResolveLabelVector(vector<long> Label_Map)

{
  unsigned long i, j;
  long label;
  vector<long>[Label_Map.size()] Help_Vector;

  j = 0;

  FOR i FROM 1 TO Label_Map.size()-1 DO
  {
    label = Label_Map[i];
    IF label == 0 THEN
    {
      j++;
      Help_Vector[i] = j;
    }
    ELSE IF Label_Map[label] <> 0 THEN
      Label_Map[i] = Label_Map[label];
  }

  FOR i FROM 1 TO Label_Map.size()-1 DO
  {
    IF Label_Map[i] == 0 THEN
      Label_Map[i] = Help_Vector[i];
    ELSE
      Label_Map[i] = Help_Vector[Label_Map[i]];
  }
}
```

The function `Update()` updates the line position according to the specified order of the lattice directions and the orientations as well as the increments for passing through the directions each time a line has been completely run length encoded.

```
void Update(vector<unsigned long>[n-1] L_Position)

{
  boolean stop_correction;
  unsigned long i;

  L_Position[0] += Increments[Rank_to_Index[1]];
```

```
  stop_correction = false;
  i = 0;

  WHILE NOT stop_correction DO
  {
    IF L_Position[i] >= Image_Size[Rank_to_Index[i+1]] THEN
    {
      L_Position[i] = 0;
      L_Position[i+1] += Increments[Rank_to_Index[i+2]];

      IF i+1 == n-2 AND L_Position[n-2] >= Image_Size[Rank_to_Index[n-1]] THEN
      {
        stop_correction = true;
        stop_main_loop = true;
      }
      i += 1;
    }
    ELSE
      stop_correction = true;
  }
}
```

The routine below computes the coordinates of a point in the current line except the coordinate in the direction with rank 0 (where runs are encoded). It is called each time the line position has been updated.

```
void Transform(vector<unsigned long>[n] Current_Pix_C,
vector<unsigned long>[n-1] L_Position)

{
  unsigned long i, j, size_j;

  FOR i FROM 1 TO n-1 DO
  {
    j = Rank_to_Index[i];
    size_j = Image_Size[j];
    Current_Pix_C[j] = (Orientations[j] == 1)? L_Position[i-1]:size_j-1-L_Position[i-1]
  }
}
```

The function `QueryLabel()` gives the label of a point by searching the array `RLE_Data` for the run comprising this point and returning its label. Since after the run length encoding runs may be filtered out which do not have the value specified by `selection_value`, if this is `<> -1`, the given point may be not contained in `RLE_Data`. In this case `-1` is returned, and the point is considered as background.

```
long QueryLabel(vector<unsigned long>[n] Point)

{
  boolean run_found;
  unsigned long i, first_direction_pos;
  long value;
  vector<unsigned long>[n-1] Line_Position;
  RLE_Line Current_RLE_Line;

  run_found = false;
  value = -1;
  first_direction_pos = Point[Rank_to_Index[0]];

  FOR i FROM 0 TO n-2 DO
    Line_Position[i] = Point[Rank_to_Index[i+1]];

  Current_RLE_Line = RLE_Data[Line_Position[n-2]][Line_Position[n-3]]...[Line_Position[0]];

  FOR i FROM 1 TO Current_RLE_Line.Runs.size() WHILE NOT run_found DO
    IF Current_RLE_Line.Runs[i-1].pos <= first_direction_pos AND >
    > first_direction_pos <= Current_RLE_Line.Runs[i-1].pos+Current_RLE_Line.Runs[i-1].length-1 THEN
    {
      run_found = true;
      value = Current_RLE_Line.Runs[i-1].label;
    }
```

```
  return(value);
}
```

6 Discussion

In this last section we discuss various aspects of our approach for encoding and labeling general image data.

Method features. The proposed labeling algorithm can be applied to images of arbitrary dimensionality and with respect to any definition of connectedness which conforms to Definition 6 in Subsection 4.2. The run length encoding and the method used to trace label correspondences make the whole algorithm quite fast and memory efficient. The tracing method resembles the technique by Thurfjell *et al.* in [21] as used in [1]. In the labeling part of our algorithm, $2m$ neighborhoods have to be tested for correspondences with m denoting the number of runs, which itself can be influenced by the choice of the scanning order. Operations and data structures whose complexity or memory demands grow exponentially in the dimensionality of the image are minimized in this way. The possibility to specify the scanning and encoding order further allows in principle to inspect any subimage and to extract the relevant data with a fast memory copy routine since for a suitable order this data is contiguous in memory.

Further applications of the run length encoding. The run length encoding as a preprocessing of the image highly facilitates the selection, filtering, and denoising of the image data to be labeled. It gives a flexible extension of morphological filtering. For example, objects whose size is below some threshold are likely to be noise. By comparing the starting points and endpoints of different runs, such objects can be filtered out before the labeling. With a proper arrangement of runs, the selection of and access to parts of the image can be sped up. Since basic object features like bounding box, volume, or barycentre of an object can be conveniently determined during the labeling by means of the run data, such features can also be utilized to appropriately organize the data for further processing. Finally, we remark that run length encoding as a simple compression can serve as a basis for more sophisticated compression methods. For large and high-dimensional images an adequate coding can be very advantageous and so should be supported by the data format in use.

Optimization and parallelization. We emphasize that, for the purpose of generality and a clear presentation, by far not all optimizations (for special cases) have been realized in our algorithm. Here we want to address some optimizations which allow a convenient extension of the code. At first, we note that special features of the neighborhood or the image, like symmetries or periodicities, might be known in advance or could be detected in an affordable manner. Their exploitation in the whole algorithm can have significant benefit in terms of memory demands and processing time. The use of tree structures and hashing methods to store and find run data with properties of special interest should also be considered. As another point we remark that the way how to build the value-to-label-map could be adjusted. In the pseudo code of `DoLabeling()`, all correspondences are collected before they are resolved in this map. In order to limit memory needs, the latter step might be performed dynamically in dependence of the number of gathered correspondences. Serializing the coding and labeling in the sense that runs are assembled only for a couple of lines and discarded after preliminary labeling enormously reduces storage needs, but also slows down the final relabeling.

The task of labeling itself as well as our implementation are very well-suited for parallelization. A simple first step towards this would be a segmentation of large images into blocks (as defined on page 7), which can be independently processed to a large extent. Such a technique is e.g. described in [13]. In this setting, of course, also the block borders have to be tested in order to completely acquire correspondences. We might gain an improved performance by processing these value equivalences in single blocks before computing the value-to-label-map for the whole image.

References

[1] A. Aguilera, J. Rodríguez, and D. Ayala. Fast connected component labeling algorithm: A non voxel-based approach. Technical report, Universitat Politècnica de Catalunya, 2002. http://www.lsi.upc.edu/dept/techreps/llistat_detallat.php?id=583.

[2] M. A. Armstrong. *Basic Topology*. Springer-Verlag, Berlin, 1997.

[3] G. Borgefors, I. Nyström, and G. Sanniti di Baja. Connected components in 3D neighbourhoods. In *Proc. 10th Scandinavian Conference on Image Analysis*, pages 567–572, Lappeenranta, Finland, 1997.

[4] J. W. S. Cassels. *An introduction to the geometry of numbers*. Springer, Berlin, Heidelberg, New York, 1971.

[5] F. Chang, C.-J. Chen, and C.-J. Lu. A linear-time component-labeling algorithm using contour tracing technique. *CVIU*, 93(2):206–220, 2004. http://www.iis.sinica.edu.tw/papers/fchang/1362-F.pdf.

[6] L. Di Stefano and A. Bulgarelli. A simple and efficient connected components labeling algorithm. In *10th International Conference on Image Analysis and Processing*, page 322, 1999. http://doi.ieeecomputersociety.org/10.1109/ICIAP.1999.797615.

[7] P. M. Gruber. Geometry of numbers. In P. M. Gruber and J. M. Wills, editors, *Handbook of convex geometry*, pages 739–763, Amsterdam, 1993. North Holland.

[8] R. Klette and A. Rosenfeld. *Digital Geometry*. Morgan & Kaufman Publ., Amsterdam, 2004.

[9] J.-O. Lachaud and A. Montanvert. Continuous analogs of digital boundaries: A topological approach to iso-surfaces. *Graphical Models*, 62:129–164, 2000.

[10] W. Nagel, J. Ohser, and K. Pischang. An integral-geometric approach for the Euler-Poincaré characteristic of spatial images. *J. Microsc.*, 198:54–62, 2000.

[11] J. Ohser, W. Nagel, and K. Schladitz. The Euler number of discretized sets – on the choice of adjacency in homogeneous lattices. In K. Mecke and D. Stoyan, editors, *Statistical Physics and Spatial Statistics*, pages 287–311, Berlin, 2002. Springer-Verlag.

[12] J. Ohser, W. Nagel, and K. Schladitz. The Euler number of discretised sets – surprising results in three dimensions. *Image Anal. Stereol.*, 22:11–19, 2003.

[13] J.-M. Park, C. G. Looney, and H.-C. Chen. Fast connected component labeling algorithm using a divide and conquer technique. Technical report, The University of Alabama, 2000. http://cs.ua.edu/research/Reports.asp.

[14] J. Rataj and M. Zähle. Nromal cycles of Lipschitz manifolds by approximation with parallel sets. *Differential Geom. Appl.*, 19:113–126, 2003.

[15] G. X. Ritter and J. N. Wilson. *Handbook of Computer Vision Algorithms in Image Algebra*, chapter 6, Connected Component Algorithms, pages 173–186. CRC Press Inc., 2001.

[16] A. Rosenfeld. Digital topology. *Amer. Math. Monthly*, 86:621–630, 1970.

[17] A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *JACM*, 13(4):471–494, 1966.

[18] J. J. Rotman. *An Introduction to Algebraic Topology*. Springer-Verlag, Berlin, 1993.

[19] K. Schladitz, J. Ohser, and W. Nagel. Measurement of intrinsic volumes of sets observed on lattices. In A. Kuba, L. G. Nyul, and K. Palagyi, editors, *The 13th International Conference on Discrete Imagery (DGCY 2006)*, 2006.

[20] M. Schmitt. Digitization and connectivity. In H. J. A. M. Heijmans and J. B. T. M. Roerdnik, editors, *Mathematical Morphology and its Application to Image and Signal Processing*, pages 91–98, Dortrecht, Boston, London, 1998. Kluwer Academic Publishers.

[21] L. Thurfjell, E. Bengtsson, and B. Nordin. A new three-dimensional connected components labeling algorithm with simultaneous object feature extraction capability. *CVGIP: Graph. Models Image Process.*, 54(4):357–364, 1992. http://portal.acm.org/citation.cfm?id=167447.167468.

IWRMM-Preprints seit 2004

Nr. 04/01 Andreas Rieder: Inexact Newton Regularization Using Conjugate Gradients as Inner IteractionMichael

Nr. 04/02 Jan Mayer: The ILUCP preconditioner

Nr. 04/03 Andreas Rieder: Runge-Kutta Integrators Yield Optimal Regularization Schemes

Nr. 04/04 Vincent Heuveline: Adaptive Finite Elements for the Steady Free Fall of a Body in a Newtonian Fluid

Nr. 05/01 Götz Alefeld, Zhengyu Wang: Verification of Solutions for Almost Linear Complementarity Problems

Nr. 05/02 Vincent Heuveline, Friedhelm Schieweck: Constrained H^1-interpolation on quadrilateral and hexahedral meshes with hanging nodes

Nr. 05/03 Michael Plum, Christian Wieners: Enclosures for variational inequalities

Nr. 05/04 Jan Mayer: ILUCDP: A Crout ILU Preconditioner with Pivoting and Row Permutation

Nr. 05/05 Reinhard Kirchner, Ulrich Kulisch: Hardware Support for Interval Arithmetic

Nr. 05/06 Jan Mayer: ILUCDP: A Multilevel Crout ILU Preconditioner with Pivoting and Row Permutation

Nr. 06/01 Willy Dörfler, Vincent Heuveline: Convergence of an adaptive *hp* finite element strategy in one dimension

Nr. 06/02 Vincent Heuveline, Hoang Nam-Dung: On two Numerical Approaches for the Boundary Control Stabilization of Semi-linear Parabolic Systems: A Comparison

Nr. 06/03 Andreas Rieder, Armin Lechleiter: Newton Regularizations for Impedance Tomography: A Numerical Study

Nr. 06/04 Götz Alefeld, Xiaojun Chen: A Regularized Projection Method for Complementarity Problems with Non-Lipschitzian Functions

Nr. 06/05 Ulrich Kulisch: Letters to the IEEE Computer Arithmetic Standards Revision Group

Nr. 06/06 Frank Strauss, Vincent Heuveline, Ben Schweizer: Existence and approximation results for shape optimization problems in rotordynamics

Nr. 06/07 Kai Sandfort, Joachim Ohser: Labeling of n-dimensional images with choosable adjacency of the pixels

Nr. 06/08 Jan Mayer: Symmetric Permutations for I-matrices to Delay and Avoid Small Pivots During Factorization

Nr. 06/09 Andreas Rieder, Arne Schneck: Optimality of the fully discrete filtered Backprojection Algorithm for Tomographic Inversion

Nr. 06/10 Patrizio Neff, Krzysztof Chelminski, Wolfgang Müller, Christian Wieners: A numerical solution method for an infinitesimal elasto-plastic Cosserat model

Nr. 06/11 Christian Wieners: Nonlinear solution methods for infinitesimal perfect plasticity

Eine aktuelle Liste aller IWRMM-Preprints finden Sie auf:

www.mathematik.uni-karlsruhe.de/iwrmm/seite/preprints