Appendix A

Mathematical Symbols and Notation

A.1 Symbols

The following symbols are used in the main text primarily with the denotations given here. While some symbols may be used for purposes other than the ones listed, the meaning should always be clear in the particular context.

$(a_0, \ldots, a_{n-1})$ A *vector* or *list*, that is, an ordered sequence of n elements of the same type. Unlike a *set* (see below), a list may contain the same element more than once. If used to denote a *vector*, then $(a_0, \ldots, a_{n-1})$ is usually a *row* vector and $(a_0, \ldots, a_{n-1})^\mathsf{T}$ is the corresponding (transposed) *column* vector.[1] If used to represent a *list*,[2] ( ) represents the *empty* list and (a) is a list with a single element a. $|A|$ is the *length* of the sequence A, that is, the number of contained elements. $A \backsim B$ denotes the concatenation of A, B. $A(i)$ or a_i refers to the i-th element of A. $A(i) \leftarrow x$ means that the i-th element of A is set to (i.e., replaced by) the quantity x.

$\{a, b, c, d, \ldots\}$ A *set*, that is, an unordered collection of distinct elements. A particular element x can be contained in a set at most once. $\{\}$ denotes the empty set. $|\mathcal{A}|$ is the size (cardinality) of the set $\mathcal{A}$. $\mathcal{A} \cup \mathcal{B}$ is the union and $\mathcal{A} \cap \mathcal{B}$ is the intersection of two sets $\mathcal{A}, \mathcal{B}$. $x \in \mathcal{A}$ means that the element x is contained in $\mathcal{A}$.

$\langle A, B, C \rangle$ A *tuple*, that is, a fixed-size, ordered sequence of elements, each possibly of a different type.[3]

[1] In most programming environments, vectors are implemented as one-dimensional arrays, with elements being referred to by position (index).

[2] Lists are usually implemented with dynamic data structures, such as linked lists. Java's *Collections* framework provides numerous easy-to-use list implementations.

[3] Tuples are typically implemented as *objects* (in Java or C++) or *structures* (in C) with elements being referred to by name.

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9

$[a, b]$	Numeric interval; $x \in [a, b]$ means $a \leq x \leq b$. Similarly, $x \in [a, b)$ says that $a \leq x < b$.
$\lvert A \rvert$	Length (number of elements) of a sequence (see above) or size (cardinality) of a set A, that is, $\lvert A \rvert \equiv \operatorname{card} A$.
$\lvert \mathbf{A} \rvert$	Determinant of a matrix $\mathbf{A}$ ($\lvert \mathbf{A} \rvert \equiv \det(\mathbf{A})$).
$\lvert x \rvert$	Absolute value (magnitude) of a scalar or complex quantity x.
$\lVert \boldsymbol{x} \rVert$	Euclidean (L_2) norm of a vector $\boldsymbol{x}$. $\lVert \boldsymbol{x} \rVert_n$ denotes the magnitude of $\boldsymbol{x}$ using a particular norm L_n.
$\lceil x \rceil$	"Ceil" of x, the smallest integer $z \in \mathbb{Z}$ greater than $x \in \mathbb{R}$. For example, $\lceil 3.141 \rceil = 4$, $\lceil -1.2 \rceil = -1$.
$\lfloor x \rfloor$	"Floor" of x, the largest integer $z \in \mathbb{Z}$ smaller than $x \in \mathbb{R}$. For example, $\lfloor 3.141 \rfloor = 3$, $\lfloor -1.2 \rfloor = -2$.
$\div$	Integer division operator: $a \div b$ denotes the quotient of the two integers a, b. For example, $5 \div 3 = 1$ and $-13 \div 4 = -3$ (equivalent to Java's "/" operator in the case of integer operands).
$*$	Linear convolution operator (see Sec. 5.3.1).
$\circledast$	Linear correlation operator (see Sec. 23.1.1).
$\otimes$	Outer vector product (see Sec. B.3.2).
$\times$	Cross product (between vectors or complex quantities (see Sec. B.3.3).
$\oplus$	Morphological dilation operator (see Sec. 9.2.3).
$\ominus$	Morphological erosion operator (see Sec. 9.2.4).
$\circ$	Morphological opening operator (see Sec. 9.3.1).
$\bullet$	Morphological closing operator (see Sec. 9.3.2).
$\smile$	Concatenation operator. Given two sequences $A = (a, b, c)$ and $B = (d, e)$, $A \smile B$ denotes the concatenation of A and B, with the result (a, b, c, d, e). Inserting a single element x at the end or front of the list A is written as $A \smile (x)$ or $(x) \smile A$, resulting in (a, b, c, x) or (x, a, b, c), respectively.
$\sim$	"Similarity" relation used in the context of random variables and statistical distributions.
$\approx$	"Approximately equal" relation.
$\equiv$	Equivalence relation.
$\leftarrow$	Assignment operator: $a \leftarrow \mathit{expr}$ means that expression *expr* is evaluated and subsequently the result is assigned to the variable a.
$\xleftarrow{+}$	Incremental assignment operator: $a \xleftarrow{+} b$ is equivalent to $a \leftarrow a + b$.
$:=$	Function definition operator (used in algorithms). For example, $f(x) := x^2 + 5$ defines a function $f()$ with the bound variable (formal function argument) x.
$\cdots$	"upto" (incrementing) iteration, used in loop constructs like **for** $q \leftarrow 1, \cdots, K$ (with $q = 1, 2, \ldots, K-1, K$).
$\cdots$	"downto" (decrementing) iteration, for example, **for** $q \leftarrow K, \cdots, 1$ (with $q = K, K-1, \ldots, 2, 1$).

$\wedge$ Logical "and" operator.

$\vee$ Logical "or" operator.

∂ Partial derivative operator (see Sec. 6.2.1). For example, $\frac{\partial}{\partial x_i} f$ denotes the *first* derivative of the multi-dimensional function $f(x_1, x_2, \ldots, x_n) : \mathbb{R}^n \to \mathbb{R}$ along variable x_i, $\frac{\partial^2}{\partial x_i^2} f$ is the *second* derivative (i.e., differentiating f twice along variable x_i), etc.

∇ Gradient operator. The gradient of a multi-dimensional function $f(x_1, x_2, \ldots, x_n) : \mathbb{R}^n \to \mathbb{R}$, denoted ∇f (also ∇_f or $\operatorname{grad} f$), is the vector of its first partial derivatives (see also Sec. C.2.2).

∇^2 Laplace operator (or *Laplacian*). The Laplacian of a multi-dimensional function $f(x_1, x_2, \ldots, x_n) : \mathbb{R}^n \to \mathbb{R}$, denoted $\nabla^2 f$ (or ∇_f^2), is the sum of its second partial derivatives (see Sec. C.2.5).

$\mathbf{0}$ Zero vector, $\mathbf{0} = (0, \ldots, 0)^\mathsf{T}$.

adj Adjugate of a square matrix, denoted $\operatorname{adj}(\mathbf{A})$; also called *adjoint* in older texts.

AND Bitwise "and" operation. Example: $(0011_\text{b}\ \text{AND}\ 1010_\text{b}) = 0010_\text{b}$ (binary) and $(3\ \text{AND}\ 6) = 2$ (decimal).

$\operatorname{ArcTan}(x, y)$ Inverse tangent function. The result of $\operatorname{ArcTan}(x, y)$ is equivalent to $\arctan(\frac{y}{x}) = \tan^{-1}(\frac{y}{x})$ but with two arguments and returning angles in the range $[-\pi, +\pi]$ (i.e., covering all four quadrants). $\operatorname{ArcTan}(x, y)$ is equivalent to the `ArcTan[x,y]` function in *Mathematica* and the `Math.atan2(y,x)` method in Java (but note the reversed arguments!).

$\mathbb{C}$ The set of complex numbers.

card Size (cardinality) of a set. $\operatorname{card}(\mathcal{A}) = |\mathcal{A}|$ (see also Sec. 3.1).

det Determinant of a matrix ($\det(\mathbf{A}) = |\mathbf{A}|$).

DFT Discrete Fourier transform (see Sec. 18.3).

e Euler's constant.

$\mathbf{e}$ Unit vector. For example, $\mathbf{e}_x = (1, 0)^\mathsf{T}$ denotes the 2D unit vector in x-direction. $\mathbf{e}_\theta = (\cos\theta, \sin\theta)^\mathsf{T}$ is the 2D unit vector oriented at angle θ and $\mathbf{e}_\text{i}, \mathbf{e}_\text{j}, \mathbf{e}_\text{k}$ are the unit vectors along the coordinate axes in 3D.

exp Exponential function: $\exp(x) = e^x$.

$\mathcal{F}$ Continuous Fourier transform (see Sec. 18.1.4).

false Boolean constant (false $= \neg$true).

grad Gradient operator (see ∇).

h Histogram of an image (see Sec. 3.1).

H Cumulative histogram (see Sec. 3.6).

$\mathbf{H}$ Hessian matrix (see Sec. C.2.6).

hom Operator for converting Cartesian to homogeneous coordinates. $\operatorname{hom}(\boldsymbol{x}) = \underline{\boldsymbol{x}}$ maps the Cartesian point $\boldsymbol{x}$ to a corresponding homogeneous point $\underline{\boldsymbol{x}}$; the reverse mapping is denoted $\operatorname{hom}^{-1}(\underline{\boldsymbol{x}}) = \boldsymbol{x}$ (see Sec. B.5).

i Imaginary unit ($\text{i}^2 = -1$), see Sec. A.3.

I	Image with scalar pixel values (e.g., an intensity or grayscale image). $I(u,v) \in \mathbb{R}$ is the pixel value at position (u,v)
$\boldsymbol{I}$	Vector-valued image, for example, a RGB color image with 3D color vectors $\boldsymbol{I}(u,v) \in \mathbb{R}^3$ at position (u,v).
$\mathbf{I}_n$	Identity matrix of size $n \times n$. For example, $\mathbf{I}_2 = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ is the 2×2 identity matrix.
$\mathbf{J}$	Jacobian matrix (see Sec. C.2.1).
$\mathrm{L}_1, \mathrm{L}_2, \mathrm{L}_\infty$	Common distance measures or *norms* (see Eqns. (15.23)–(15.25)).
$M \times N$	Domain of pixel coordinates (u,v) for an image with M columns (width) and N rows (height); used as a shortcut notation for the set $\{0,\ldots,M-1\} \times \{0,\ldots,N-1\}$.
mod	Modulus operator: $(a \bmod b)$ is the remainder of the *integer* division $a \div b$ (see Sec. F.1.2).
μ	Arithmetic mean value.
$\mathbb{N}$	The set of natural numbers; $\mathbb{N} = \{1,2,3,\ldots\}$, $\mathbb{N}_0 = \{0,1,2,\ldots\}$.
nil	Null ("nothing") constant, typically used in algorithms to denote an invalid quantity (similar to `null` in Java).
p	Discrete probability density function (see Sec. 4.6.1).
P	Discrete probability distribution function or cumulative probability density (see Sec. 4.6.1).
$\mathcal{Q}$	Quadrilateral (see Sec. 21.1.4).
$\mathbb{R}$	The set of real numbers.
R, G, B	*Red*, *green* and *blue* color components.
rank	Rank of a matrix $\mathbf{A}$, denoted by $\mathrm{rank}(\mathbf{A})$.
round	Rounding function: returns the integer closest to the scalar $x \in \mathbb{R}$. $\mathrm{round}(x) \equiv \lfloor x + 0.5 \rfloor$.
σ	Standard deviation (square root of the *variance* σ^2).
$\mathcal{S}_1$	Unit square (see Sec. 21.1.4).
sgn	"Sign" or "signum" function: $\mathrm{sgn}(x) = \begin{cases} 1 & \text{for } x > 0 \\ 0 & \text{for } x = 0 \\ -1 & \text{for } x < 0 \end{cases}$
τ	Interval in time or space.
t	Continuous time variable.
t	Threshold value.
T	Transpose of a vector ($\boldsymbol{a}^\mathsf{T}$) or matrix ($\mathbf{A}^\mathsf{T}$).
trace	Trace (sum of the diagonal elements) of a matrix, e.g., $\mathrm{trace}(\mathbf{A})$.
true	Boolean constant (true = ¬false).
$\boldsymbol{u} = (u,v)$	Discrete 2D coordinate variable with $u, v \in \mathbb{Z}$.
$\boldsymbol{x} = (x,y)$	Continuous 2D coordinate variable with $x, y \in \mathbb{R}$.
XOR	Bitwise "xor" (exclusive OR) operator. Example: $(0011_\mathrm{b}$ XOR $1010_\mathrm{b}) = 1001_\mathrm{b}$ (binary) and $(3 \text{ XOR } 6) = 5$ (decimal).
$\mathbb{Z}$	The set of integers.

A.2 Set Operators

$\lvert\mathcal{A}\rvert$	The size of the set $\mathcal{A}$ (equal to card($\mathcal{A}$)).
$\forall_x \ldots$	"All" quantifier (for all x, ...).
$\exists_x \ldots$	"Exists" quantifier (there is some x for which ...).
$\cup$	Set union (e.g., $\mathcal{A} \cup \mathcal{B}$).
$\cap$	Set intersection (e.g., $\mathcal{A} \cap \mathcal{B}$).
$\bigcup_i \mathcal{A}_i$	Union of multiple sets $\mathcal{A}_i$.
$\bigcap_i \mathcal{A}_i$	Intersection over multiple sets $\mathcal{A}_i$.
$\setminus$	Set difference: if $x \in \mathcal{A} \setminus \mathcal{B}$, then $x \in \mathcal{A}$ and $x \notin \mathcal{B}$.

A.3 Complex Numbers

Basic relations:

$$z = a + \mathrm{i} \cdot b \qquad (\text{with } z, \mathrm{i} \in \mathbb{C},\ a, b \in \mathbb{R},\ \mathrm{i}^2 = -1) \tag{A.1}$$

$$s \cdot z = s \cdot a + \mathrm{i} \cdot s \cdot b \qquad (\text{for } s \in \mathbb{R}) \tag{A.2}$$

$$|z| = \sqrt{a^2 + b^2} \tag{A.3}$$

$$|s \cdot z| = s \cdot |z| \tag{A.4}$$

$$z = a + \mathrm{i} \cdot b = |z| \cdot (\cos\psi + \mathrm{i} \cdot \sin\psi) \tag{A.5}$$

$$= |z| \cdot e^{\mathrm{i}\cdot\psi} \qquad (\text{with } \psi = \mathrm{ArcTan}(a, b)) \tag{A.6}$$

$$\mathrm{Re}\big(a + \mathrm{i} \cdot b\big) = a \qquad \mathrm{Re}\big(e^{\mathrm{i}\cdot\varphi}\big) = \cos\varphi \tag{A.7}$$

$$\mathrm{Im}\big(a + \mathrm{i} \cdot b\big) = b \qquad \mathrm{Im}\big(e^{\mathrm{i}\cdot\varphi}\big) = \sin\varphi \tag{A.8}$$

$$e^{\mathrm{i}\cdot\varphi} = \cos\varphi + \mathrm{i} \cdot \sin\varphi \tag{A.9}$$

$$e^{-\mathrm{i}\cdot\varphi} = \cos\varphi - \mathrm{i} \cdot \sin\varphi \tag{A.10}$$

$$\cos(\varphi) = \tfrac{1}{2} \cdot \big(e^{\mathrm{i}\cdot\varphi} + e^{-\mathrm{i}\cdot\varphi}\big) \tag{A.11}$$

$$\sin(\varphi) = \tfrac{1}{2i} \cdot \big(e^{\mathrm{i}\cdot\varphi} - e^{-\mathrm{i}\cdot\varphi}\big) \tag{A.12}$$

$$z^* = a - \mathrm{i} \cdot b \qquad (\text{complex conjugate}) \tag{A.13}$$

$$z \cdot z^* = z^* \cdot z = |z|^2 = a^2 + b^2 \tag{A.14}$$

$$z^0 = (a + \mathrm{i} \cdot b)^0 = (1 + \mathrm{i} \cdot 0) = 1 \tag{A.15}$$

Arithmetic operations:

$$z_1 = (a_1 + \mathrm{i} \cdot b_1) = |z_1|\, e^{\mathrm{i}\cdot\varphi_1} \tag{A.16}$$

$$z_2 = (a_2 + \mathrm{i} \cdot b_2) = |z_2|\, e^{\mathrm{i}\cdot\varphi_2} \tag{A.17}$$

$$z_1 + z_2 = (a_1 + a_2) + \mathrm{i} \cdot (b_1 + b_2), \tag{A.18}$$

$$z_1 \cdot z_2 = (a_1 \cdot a_2 - b_1 \cdot b_2) + \mathrm{i} \cdot (a_1 \cdot b_2 + b_1 \cdot a_2) \tag{A.19}$$

$$= |z_1| \cdot |z_2| \cdot e^{\mathrm{i}\cdot(\varphi_1 + \varphi_2)} \tag{A.20}$$

$$\frac{z_1}{z_2} = \frac{a_1 \cdot a_2 + b_1 \cdot b_2}{a_2^2 + b_2^2} + \mathrm{i} \cdot \frac{a_2 \cdot b_1 - a_1 \cdot b_2}{a_2^2 + b_2^2} = \frac{|z_1|}{|z_2|} \cdot e^{\mathrm{i}\cdot(\varphi_1 - \varphi_2)} \tag{A.21}$$

Appendix B

Linear Algebra

This part contains a compact set of elementary tools and concepts from algebra and calculus that are referenced in the main text. Many good textbooks (probably including some of your school books) are available on this subject, for example, [35,36,145,264]. For numerical aspects of linear algebra see [160,190].

B.1 Vectors and Matrices

Here we describe the basic notation for vectors in two and three dimensions. Let

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \tag{B.1}$$

denote vectors $\boldsymbol{a}, \boldsymbol{b}$ in 2D, and analogously

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} \tag{B.2}$$

vectors in 3D (with $a_i, b_i \in \mathbb{R}$). Vectors are used to describe 2D or 3D points (relative to the origin of the coordinate system) or the displacement between two arbitrary points in the corresponding space.

We commonly use upper-case letters to denote a *matrix*, for example,

$$\mathbf{A} = \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \\ A_{2,0} & A_{2,1} \end{pmatrix}. \tag{B.3}$$

This matrix consists of 3 rows and 2 columns; in other words, $\mathbf{A}$ is of size $(3, 2)$. Its individual elements are referenced as $A_{i,j}$, where i is the *row* index (vertical coordinate) and j is the *column* index (horizontal coordinate).[1]

[1] Note that the usual notation for matrix coordinates is (unlike image coordinates) vertical-first!

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9

The *transpose* of $\mathbf{A}$, denoted $\mathbf{A}^\mathsf{T}$, is obtained be exchanging rows and columns, that is,

$$\mathbf{A}^\mathsf{T} = \begin{pmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \\ A_{2,0} & A_{2,1} \end{pmatrix}^\mathsf{T} = \begin{pmatrix} A_{0,0} & A_{1,0} & A_{2,0} \\ A_{0,1} & A_{1,1} & A_{2,1} \end{pmatrix}. \tag{B.4}$$

The *inverse* of a square matrix $\mathbf{A}$ is denoted $\mathbf{A}^{-1}$, such that

$$\mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I} \qquad \text{and} \qquad \mathbf{A}^{-1} \cdot \mathbf{A} = \mathbf{I} \tag{B.5}$$

($\mathbf{I}$ is the identity matrix). Note that not every square matrix has an inverse. Calculation of the inverse can be performed in closed form up to the size $(3, 3)$; for example, see Eqn. (21.29) and Eqn. (24.47). In general, the use of standard numerical methods is recommended (see Sec. B.6).

B.1.1 Column and Row Vectors

For practical purposes, a vector can be considered a special case of a matrix. In particular, a the m-dimensional *column* vector

$$\boldsymbol{a} = \begin{pmatrix} a_0 \\ \vdots \\ a_{m-1} \end{pmatrix} \tag{B.6}$$

corresponds to a matrix of size $(m, 1)$, while its transpose $\boldsymbol{a}^\mathsf{T}$ is a *row* vector and thus like a matrix of size $(1, m)$. By default, and unless otherwise noted, any vector is implicitly assumed to be a *column* vector.

B.1.2 Length (Norm) of a Vector

The *length* or *Euclidean norm* (L_2 norm) of a vector $\boldsymbol{a} = (a_1, \ldots, a_{m-1})^\mathsf{T}$, denoted $\|\boldsymbol{a}\|$, is defined as

$$\|\boldsymbol{a}\| = \Big(\sum_{i=0}^{m-1} a_i^2\Big)^{1/2}. \tag{B.7}$$

For example, the length of the 3D vector $\boldsymbol{x} = (x, y, z)^\mathsf{T}$ is

$$\|\boldsymbol{x}\| = \sqrt{x^2 + y^2 + z^2}. \tag{B.8}$$

B.2 Matrix Multiplication

B.2.1 Scalar Multiplication

The product of a real-valued matrix and a scalar value $s \in \mathbb{R}$ is defined as

$$s \cdot \mathbf{A} = \mathbf{A} \cdot s = \big[s \cdot A_{i,j}\big] = \begin{pmatrix} s \cdot A_{0,0} & \cdots & s \cdot A_{0,n-1} \\ \vdots & \ddots & \vdots \\ s \cdot A_{m-1,0} & \cdots & s \cdot A_{m-1,n-1} \end{pmatrix}. \tag{B.9}$$

B.2.2 Product of Two Matrices

We say that a matrix is of size (m, n) if consists of m rows and n columns. Given two matrices $\mathbf{A}$, $\mathbf{B}$ of size (m, n) and (p, q), respectively, the product $\mathbf{A} \cdot \mathbf{B}$ is only defined if $n = p$. Thus the number of columns (n) in $\mathbf{A}$ must always match the number of rows (p) in $\mathbf{B}$. The result is a new matrix $\mathbf{C}$ of size (m, q), that is,

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \underbrace{\begin{pmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{pmatrix}}_{(m,n)} \cdot \underbrace{\begin{pmatrix} B_{0,0} & \cdots & B_{0,q-1} \\ \vdots & \ddots & \vdots \\ B_{n-1,0} & \cdots & B_{n-1,q-1} \end{pmatrix}}_{(n,q)}$$
$$= \underbrace{\begin{pmatrix} C_{0,0} & \cdots & C_{0,q-1} \\ \vdots & \ddots & \vdots \\ C_{m-1,0} & \cdots & C_{m-1,q-1} \end{pmatrix}}_{(m,q)}, \tag{B.10}$$

with the elements

$$C_{ij} = \sum_{k=0}^{n-1} A_{i,k} \cdot B_{k,j}, \tag{B.11}$$

for $i = 0, \ldots, m-1$ and $j = 0, \ldots, q-1$. Note that this product is not commutative, that is, $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A}$ in general.

B.2.3 Matrix-Vector Products

The product $\mathbf{A} \cdot \boldsymbol{x}$ between a matrix $\mathbf{A}$ and a vector $\boldsymbol{x}$ is only a special case of the matrix-matrix multiplication given in Eqn. (B.10). In particular, if $\boldsymbol{x} = (x_0, \ldots, x_{n-1})^\mathsf{T}$ is a n-dimensional *column* vector (i.e., a matrix of size $(n, 1)$), then the multiplication

$$\underbrace{\boldsymbol{y}}_{(m,1)} = \underbrace{\mathbf{A}}_{(m,n)} \cdot \underbrace{\boldsymbol{x}}_{(n,1)} \tag{B.12}$$

is only defined if the matrix $\mathbf{A}$ is of size (m, n), for arbitrary $m \geq 1$. The result $\boldsymbol{y}$ is a *column* vector of length m (equivalent to a matrix of size $(m, 1)$). For example (with $m = 2$, $n = 3$),

$$\mathbf{A} \cdot \boldsymbol{x} = \underbrace{\begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix}}_{(2,3)} \cdot \underbrace{\begin{pmatrix} x \\ y \\ z \end{pmatrix}}_{(3,1)} = \underbrace{\begin{pmatrix} A \cdot x + B \cdot y + C \cdot z \\ D \cdot x + E \cdot y + F \cdot z \end{pmatrix}}_{(2,1)}. \tag{B.13}$$

Here $\mathbf{A}$ operates on the column vector $\boldsymbol{x}$ "from the left", that is, $\mathbf{A} \cdot \boldsymbol{x}$ is the *left-sided* matrix-vector product of $\mathbf{A}$ and $\boldsymbol{x}$.

Similarly, a *right-sided* multiplication of a *row* vector $\boldsymbol{x}^\mathsf{T}$ of length m with a matrix of size (m, n) is performed as

$$\underbrace{\boldsymbol{x}^\mathsf{T}}_{(1,m)} \cdot \underbrace{\mathbf{B}}_{(m,n)} = \underbrace{\boldsymbol{z}}_{(1,n)}, \tag{B.14}$$

where the result $\boldsymbol{z}$ is a n-dimensional *row* vector; for example (again with $m = 2$, $n = 3$),

$$\boldsymbol{x}^\mathsf{T} \cdot \mathbf{B} = \underbrace{(x, y)}_{(1,2)} \cdot \underbrace{\begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix}}_{(2,3)} = \underbrace{\big(x \cdot A + y \cdot D,\; x \cdot B + y \cdot E,\; x \cdot C + y \cdot F\big)}_{(1,3)}. \tag{B.15}$$

In general, if $\mathbf{A} \cdot \boldsymbol{x}$ is defined, then

$$\mathbf{A} \cdot \boldsymbol{x} = (\boldsymbol{x}^\mathsf{T} \cdot \mathbf{A}^\mathsf{T})^\mathsf{T} \qquad \text{and} \qquad (\mathbf{A} \cdot \boldsymbol{x})^\mathsf{T} = \boldsymbol{x}^\mathsf{T} \cdot \mathbf{A}^\mathsf{T}. \tag{B.16}$$

Thus, any right-sided matrix-vector product $\mathbf{A} \cdot \boldsymbol{x}$ can also be calculated as a left-sided product $\boldsymbol{x}^\mathsf{T} \cdot \mathbf{A}^\mathsf{T}$ by transposing the corresponding matrix $\mathbf{A}$ and vector $\boldsymbol{x}$.

B.3 Vector Products

Products between vectors are a common cause of confusion, mainly because the same symbol $(\cdot)$ is used to denote widely different operators.

B.3.1 Dot (Scalar) Product

The *dot* product (also called *scalar* or *inner* product) of two vectors $\boldsymbol{a} = (a_0, \ldots, a_{n-1})^\mathsf{T}$, $\boldsymbol{b} = (b_0, \ldots, b_{n-1})^\mathsf{T}$ of the same length n is defined as

$$x = \boldsymbol{a} \cdot \boldsymbol{b} = \sum_{i=0}^{n-1} a_i \cdot b_i. \tag{B.17}$$

Thus the result x is a scalar value (hence the name of this product). If we write this as the product of a row and a column vector, as in Eqn. (B.14),

$$\underbrace{x}_{(1,1)} = \underbrace{\boldsymbol{a}^\mathsf{T}}_{(1,n)} \cdot \underbrace{\boldsymbol{b}}_{(n,1)}, \tag{B.18}$$

we conclude that the result x is a matrix of size $(1, 1)$, that is, a single scalar value. The dot product can be viewed as the *projection* of one vector onto the other, with the relation

$$\boldsymbol{a} \cdot \boldsymbol{b} = \|\boldsymbol{a}\| \cdot \|\boldsymbol{b}\| \cdot \cos(\alpha), \tag{B.19}$$

where α is angle enclosed by the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$. As a consequence, the dot product is *zero* if the two vectors are *orthogonal* to each other.

The dot product of a vector with *itself* gives the square of its length (see Eqn. (B.7)), that is,

$$\boldsymbol{a} \cdot \boldsymbol{a} = \sum_{i=0}^{n-1} a_i^2 = \|\boldsymbol{a}\|^2. \tag{B.20}$$

B.3.2 Outer Product

The outer product of two vectors $\boldsymbol{a} = (a_0, \ldots, a_{m-1})^\mathsf{T}$, $\boldsymbol{b} = (b_0, \ldots, b_{n-1})^\mathsf{T}$ of length m and n, respectively, is defined as

$$\mathbf{M} = \boldsymbol{a} \otimes \boldsymbol{b} = \boldsymbol{a} \cdot \boldsymbol{b}^\mathsf{T} = \begin{pmatrix} a_0 b_0 & a_0 b_1 & \ldots & a_0 b_{n-1} \\ a_1 b_0 & a_1 b_1 & \ldots & a_1 b_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1} b_0 & a_{m-1} b_1 & \ldots & a_{m-1} b_{n-1} \end{pmatrix}. \quad \text{(B.21)}$$

Thus the result is a *matrix* $\mathbf{M}$ with m rows and n columns and elements $M_{ij} = a_i \cdot b_j$, for $i = 0, \ldots, m-1$ and $j = 1, \ldots, n-1$. Note that $\boldsymbol{a} \cdot \boldsymbol{b}^\mathsf{T}$ in Eqn. (B.21) denotes the ordinary (matrix) product of the column vector $\boldsymbol{a}$ (of size $m \times 1$) and the row vector $\boldsymbol{b}^\mathsf{T}$ (of size $1 \times n$), as defined in Eqn. (B.10). The outer product is a special case of the *Kronecker* product ($\otimes$) which generally operates on pairs of matrices.

B.3.3 Cross Product

Although the cross product ($\times$) is generally defined for n-dimensional vectors, it is almost exclusively used in the 3D case, where the result is geometrically easy to understand. For a pair of 3D vectors, $\boldsymbol{a} = (a_0, a_1, a_2)^\mathsf{T}$ and $\boldsymbol{b} = (b_0, b_1, b_2)^\mathsf{T}$, the *cross product* is defined as

$$\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} \times \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} a_1 \cdot b_2 - a_2 \cdot b_1 \\ a_2 \cdot b_0 - a_0 \cdot b_2 \\ a_0 \cdot b_1 - a_1 \cdot b_0 \end{pmatrix}. \quad \text{(B.22)}$$

In the 3D case, the *cross product* is another 3D vector that is perpendicular to both of the original vectors.[2] The magnitude (length) of the vector $\boldsymbol{c}$ relates to the angle θ between $\boldsymbol{a}$ and $\boldsymbol{b}$ as

$$\|\boldsymbol{c}\| = \|\boldsymbol{a} \times \boldsymbol{b}\| = \|\boldsymbol{a}\| \cdot \|\boldsymbol{b}\| \cdot \sin(\theta). \quad \text{(B.23)}$$

The quantity $\|\boldsymbol{a} \times \boldsymbol{b}\|$ corresponds to the area of the parallelogram spanned by the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$.

B.4 Eigenvectors and Eigenvalues

This section gives an elementary introduction to eigenvectors and eigenvalues, which are mentioned at several places in the main text (see also [27, 64]). In general, the eigenvalue problem is to find solutions $\boldsymbol{x} \in \mathbb{R}^n$ and $\lambda \in \mathbb{R}$ for the linear equation

$$\mathbf{A} \cdot \boldsymbol{x} = \lambda \cdot \boldsymbol{x}\,, \quad \text{(B.24)}$$

with the given square matrix $\mathbf{A}$ of size (n, n). Any non-trivial[3] solution $\boldsymbol{x}$ is an *eigenvector* of $\mathbf{A}$ and the scalar λ (which may be

[2] For dimensions greater than three, the definition (and calculation) of the cross product is considerably more involved.

[3] An obvious but trivial solution is $\boldsymbol{x} = \mathbf{0}$ (where $\mathbf{0}$ denotes the zero-vector).

complex-valued) is the associated *eigenvalue*. Eigenvalue and eigenvectors thus always come in pairs $\langle \lambda_j, \boldsymbol{x}_j \rangle$, usually called *eigenpairs*. Geometrically speaking, applying the matrix $\mathbf{A}$ to an eigenvector only changes the vector's *magnitude* or *length* (by the associated eigenvalue λ), but not its orientation in space. Equation (B.24) can be rewritten as

$$\mathbf{A} \cdot \boldsymbol{x} - \lambda \cdot \boldsymbol{x} = \mathbf{0} \qquad \text{or} \qquad \big(\mathbf{A} - \lambda \cdot \mathbf{I}_n\big) \cdot \boldsymbol{x} = \mathbf{0}\,, \tag{B.25}$$

where $\mathbf{I}_n$ is the (n, n) identity matrix. This homogeneous linear equation has non-trivial solutions only if the matrix $(\mathbf{A} - \lambda \cdot \mathbf{I}_n)$ is *singular*, that is, its rank is *less* than n and thus its determinant det() is zero, that is,

$$\det\big(\mathbf{A} - \lambda \cdot \mathbf{I}_n\big) = 0\,. \tag{B.26}$$

Equation (B.26) is called the "characteristic equation" of the matrix $\mathbf{A}$ and can be expanded to a n-th order polynomial in λ. This polynomial has a maximum of n distinct roots, which are the eigenvalues of $\mathbf{A}$ (that is, solutions to Eqn. (B.26)). A matrix of size (n, n) thus has up to n non-distinct eigenvectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$, each with an associated eigenvalue $\lambda_1, \lambda_2, \ldots, \lambda_n$.

If they exist, the eigen*values* of a matrix are *unique*, but the associated eigen*vectors* are not! This results from the fact that, if Eqn. (B.24) is satisfied for a vector $\boldsymbol{x}$ (and the associated eigenvalue λ), it also applies to any *scaled* vector $s\boldsymbol{x}$, that is,

$$\mathbf{A} \cdot s\boldsymbol{x} = \lambda \cdot s\boldsymbol{x}\,, \tag{B.27}$$

for arbitrary $s \in \mathbb{R}$ (and $s \neq 0$). Thus, if $\boldsymbol{x}$ is an eigenvector of $\mathbf{A}$, then $s\boldsymbol{x}$ is also an (equivalent) eigenvector.

Note that the eigenvalues of a real-valued matrix may generally be complex. However, (as an important special case) if the matrix $\mathbf{A}$ is *real* and *symmetric*, *all* its eigenvalues are guaranteed to be *real*.

Example

For the real-valued (non-symmetric) 2×2 matrix

$$\mathbf{A} = \begin{pmatrix} 3 & -2 \\ -4 & 1 \end{pmatrix},$$

the two eigenvalues and their associated eigenvectors are

$$\lambda_1 = 5, \quad \boldsymbol{x}_1 = s \cdot \begin{pmatrix} 4 \\ -4 \end{pmatrix}, \qquad \text{and} \qquad \lambda_2 = -1, \quad \boldsymbol{x}_2 = s \cdot \begin{pmatrix} -2 \\ -4 \end{pmatrix},$$

for any nonzero $s \in \mathbb{R}$. The result can be easily verified by inserting pairs $\langle \lambda_1, \boldsymbol{x}_1 \rangle$ and $\langle \lambda_2, \boldsymbol{x}_2 \rangle$, respectively, into Eqn. (B.24).

B.4.1 Calculation of Eigenvalues

Special case: 2×2 matrix

For the special (but frequent) case of $n = 2$, the solution can be found in closed form (and without any software libraries). In this case, the characteristic equation (Eqn. (B.26)) reduces to

Alg. B.1
Calculating the real eigenvalues and eigenvectors for a 2×2 real-valued matrix $\mathbf{A}$. If the matrix has real eigenvalues, an ordered sequence of two "eigenpairs" $\langle \lambda_i, \boldsymbol{x}_i \rangle$, each containing the eigenvalue λ_i and the associated eigenvector $\boldsymbol{x}_i$, is returned ($i = 1, 2$). The resulting sequence is ordered by decreasing eigenvalues. nil is returned if $\mathbf{A}$ has no real eigenvalues.

RealEigenValues2x2 (A, B, C, D)
Input: $A, B, C, D \in \mathbb{R}$, the elements of a real-valued 2×2 matrix $\mathbf{A} = \left(\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right)$. Returns an ordered sequence of real-valued eigenpairs $\langle \lambda_i, \boldsymbol{x}_i \rangle$ for $\mathbf{A}$, or nil if the matrix has no real-valued eigenvalues.

$R \leftarrow \frac{A+D}{2}$
$S \leftarrow \frac{A-D}{2}$
if $(S^2 + B \cdot C) < 0$ **then**
  return nil ▷ $\mathbf{A}$ has no real-valued eigenvalues
else
  $T \leftarrow \sqrt{S^2 + B \cdot C}$
  $\lambda_1 \leftarrow R + T$ ▷ eigenvalue λ_1
  $\lambda_2 \leftarrow R - T$ ▷ eigenvalue λ_2
  if $(A - D) \geq 0$ **then**
    $\boldsymbol{x}_1 \leftarrow (S + T, C)^\intercal$ ▷ eigenvector $\boldsymbol{x}_1$
    $\boldsymbol{x}_2 \leftarrow (B, -S - T)^\intercal$ ▷ eigenvector $\boldsymbol{x}_2$
  else
    $\boldsymbol{x}_1 \leftarrow (B, -S + T)^\intercal$ ▷ eigenvector $\boldsymbol{x}_1$
    $\boldsymbol{x}_2 \leftarrow (S - T, C)^\intercal$ ▷ eigenvector $\boldsymbol{x}_2$
  return $(\langle \lambda_1, \boldsymbol{x}_1 \rangle, \langle \lambda_2, \boldsymbol{x}_2 \rangle)$ ▷ $\lambda_1 \geq \lambda_2$

$$\det\big(\mathbf{A} - \lambda \cdot \mathbf{I}_2\big) = \left| \begin{pmatrix} A & B \\ C & D \end{pmatrix} - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right| = \begin{vmatrix} A-\lambda & B \\ C & D-\lambda \end{vmatrix} \tag{B.28}$$

$$= \lambda^2 - (A + D) \cdot \lambda + (AD - BC) = 0\,. \tag{B.29}$$

The two possible solutions to this quadratic equation,

$$\begin{aligned} \lambda_{1,2} &= \frac{A+D}{2} \pm \left[\left(\frac{A+D}{2} \right)^2 - (AD - BC) \right]^{1/2} \\ &= \frac{A+D}{2} \pm \left[\left(\frac{A-D}{2} \right)^2 + BC \right]^{1/2} \\ &= R \pm \sqrt{S^2 + BC}, \end{aligned} \tag{B.30}$$

are the eigenvalues of the matrix $\mathbf{A}$, with

$$\begin{aligned} \lambda_1 &= R + \sqrt{S^2 + B \cdot C}, \\ \lambda_2 &= R - \sqrt{S^2 + B \cdot C}. \end{aligned} \tag{B.31}$$

Both λ_1, λ_2 are real-valued if the term under the square root is positive, that is, if

$$S^2 + B \cdot C = \left(\frac{A-D}{2} \right)^2 + B \cdot C \;\geq\; 0\,. \tag{B.32}$$

In particular, if the matrix is *symmetric* (i.e., $B = C$), this condition is guaranteed (because $B \cdot C \geq 0$). In this case, $\lambda_1 \geq \lambda_2$. Algorithm B.1[4] summarizes the closed-form computation of the eigenvalues and eigenvectors of a 2×2 matrix.

[4] See [27] and its reprint in [28, Ch. 5].

General case: $n \times n$

In general, proven numerical software should be used for eigenvalue calculations. See the example using the Apache Commons Math library in Sec. B.6.5.

B.5 Homogeneous Coordinates

Homogeneous coordinates are an alternative representation of points in multi-dimensional space. They are commonly used in 2D and 3D geometry because they can greatly simplify the description of certain transformations. For example, affine and projective transformations become matrices with homogeneous coordinates and the composition of transformations can be performed by simple matrix multiplication.[5]

To convert a given n-dimensional *Cartesian* point $\boldsymbol{x} = (x_0, \ldots, x_{n-1})^\mathsf{T}$ to *homogeneous* coordinates $\underline{\boldsymbol{x}}$, we use the notation[6]

$$\hom(\boldsymbol{x}) = \underline{\boldsymbol{x}}. \tag{B.33}$$

This operation increases the dimensionality of the original vector by one by inserting the additional element 1, that is,

$$\hom \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \\ 1 \end{pmatrix} = \begin{pmatrix} \underline{x}_0 \\ \vdots \\ \underline{x}_{n-1} \\ \underline{x}_n \end{pmatrix}. \tag{B.34}$$

Note that the homogeneous representation of a Cartesian vector is not unique, but every multiple of the homogeneous vector is an equivalent representation of $\boldsymbol{x}$. Thus any scaled homogeneous vector $\underline{\boldsymbol{x}}' = s \cdot \underline{\boldsymbol{x}}$ (with $s \in \mathbb{R}$, $s \neq 0$) corresponds to the *same* Cartesian vector (see also Eqn. (B.39)).

To convert a given homogeneous point $\underline{\boldsymbol{x}} = (\underline{x}_0, \ldots, \underline{x}_n)^\mathsf{T}$ back to Cartesian coordinates $\boldsymbol{x}$ we simply write

$$\hom^{-1}(\underline{\boldsymbol{x}}) = \boldsymbol{x}. \tag{B.35}$$

This operation can be easily derived as

$$\hom^{-1} \begin{pmatrix} \underline{x}_0 \\ \vdots \\ \underline{x}_{n-1} \\ \underline{x}_n \end{pmatrix} = \frac{1}{\underline{x}_n} \cdot \begin{pmatrix} \underline{x}_0 \\ \vdots \\ \underline{x}_{n-1} \end{pmatrix} = \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix}, \tag{B.36}$$

provided that $\underline{x}_n \neq 0$. Two homogeneous points $\underline{\boldsymbol{x}}_1, \underline{\boldsymbol{x}}_2$ are considered *equivalent* ($\equiv$), if they represent the same Cartesian point, that is,

$$\underline{\boldsymbol{x}}_1 \equiv \underline{\boldsymbol{x}}_2 \quad \Leftrightarrow \quad \hom^{-1}(\underline{\boldsymbol{x}}_1) = \hom^{-1}(\underline{\boldsymbol{x}}_2). \tag{B.37}$$

It follows from Eqn. (B.36) that

[5] See Chapter 21, Sec. 21.1.2.
[6] The operator hom( ) is introduced here for convenience and clarity.

$$\hom^{-1}(\underline{\boldsymbol{x}}) = \hom^{-1}(s \cdot \underline{\boldsymbol{x}}) \tag{B.38}$$

for any nonzero factor $s \in \mathbb{R}$. Thus, as mentioned earlier, any scaled homogeneous point corresponds to the same Cartesian point, that is,

$$\underline{\boldsymbol{x}} \equiv s \cdot \underline{\boldsymbol{x}}. \tag{B.39}$$

For example, for the Cartesian point $\boldsymbol{x} = (3, 7, 2)^\mathsf{T}$, the homogeneous coordinates

$$\hom(\boldsymbol{x}) = \begin{pmatrix} 3 \\ 7 \\ 2 \\ 1 \end{pmatrix} \equiv \begin{pmatrix} -3 \\ -7 \\ -2 \\ -1 \end{pmatrix} \equiv \begin{pmatrix} 9 \\ 31 \\ 6 \\ 3 \end{pmatrix} \equiv \begin{pmatrix} 30 \\ 70 \\ 20 \\ 10 \end{pmatrix} \dots \tag{B.40}$$

are all equivalent. Homogeneous coordinates can be used for vector spaces of arbitrary dimension, including 2D coordinates.

B.6 Basic Matrix-Vector Operations with the *Apache Commons Math* Library

It is recommended to use proven standard software, such as the *Apache Commons Math*[7] (ACM) library, for any non-trivial linear algebra calculation.

B.6.1 Vectors and Matrices

The basic data structures for representing vectors and matrices are `RealVector` and `RealMatrix`, respectively. The following ACM examples show the conversion from and to simple Java arrays of element-type `double`:

```
import org.apache.commons.math3.linear.MatrixUtils;
import org.apache.commons.math3.linear.RealMatrix;
import org.apache.commons.math3.linear.RealVector;

// Data given as simple arrays:
double[]   xa = {1, 2, 3};
double[][] Aa = {{2, 0, 1}, {0, 2, 0}, {1, 0, 2}};

// Conversion to vectors and matrices:
RealVector x = MatrixUtils.createRealVector(xa);
RealMatrix A = MatrixUtils.createRealMatrix(Aa);

// Get a single matrix element A_{i,j}:
int i, j; // specify row (i) and column (j)
double aij = A.getEntry(i, j);

// Set a single matrix element to a new value:
double value;
A.setEntry(i, j, value);

// Extract data to arrays again:
double[]   xb = x.toArray();
double[][] Ab = A.getData();
```

[7] http://commons.apache.org/math/.

```
// Transpose the matrix A:
RealMatrix At = A.transpose();
```

B.6.2 Matrix-Vector Multiplication

The following examples show how to implement the various matrix-vector products described in Sec. B.2.3.

```
RealMatrix A = ...; // matrix A of size (m, n)
RealMatrix B = ...; // matrix B of size (p, q), with p = n
RealVector x = ...; // vector x of length n

// Scalar multiplication C ← s · A:
double s = ...;
RealMatrix C = A.scalarMultiply(s);

// Product of two matrices: C ← A · B:
RealMatrix C = A.multiply(B); // C is of size (m, q)

// Left-sided matrix-vector product: y ← A · x:
RealVector y = A.operate(x);

// Right-sided matrix-vector product: y ← x^T · A:
RealVector y = A.preMultiply(x);
```

B.6.3 Vector Products

The following code segments show the use of the ACM library for calculating various vector products described in Sec. B.3.

```
RealVector a, b;   // vectors a, b (both of length n)

// Multiplication by a scalar c ← s · a:
double s;
RealVector c = a.mapMultiply(s);

// Dot (scalar) product x ← a · b:
double x = a.dotProduct(b);

// Outer product M ← a ⊗ b:
RealMatrix M = a.outerProduct(b);
```

B.6.4 Inverse of a Square Matrix

The following example shows the inversion of a square matrix:

```
RealMatrix A  = ... ;    // a square matrix
RealMatrix Ai = MatrixUtils.inverse(A);
```

B.6.5 Eigenvalues and Eigenvectors

The following code segment illustrates the calculation of eigenvalues and eigenvalues of a square matrix `A` using the class `EigenDecomposition` of the Apache Commons Math API. Note that the eigenval-

ues returned by `getRealEigenvalues()` are sorted in non-increasing order. The same ordering applies to the associated eigenvectors.

```
import org.apache.commons.math3.linear.EigenDecomposition;
...

RealMatrix A = MatrixUtils.createRealMatrix(new double[][]
    {{2, 0, 1},
     {0, 2, 0},
     {1, 0, 2}});

EigenDecomposition ed = new EigenDecomposition(A);

if (ed.hasComplexEigenvalues()) {
  System.out.println("A has complex Eigenvalues!");
}
else {
  // get all real eigenvalues:
  double[] lambda = ed.getRealEigenvalues();   // = (3, 2, 1)
  // get the associated eigenvectors:
  for (int i = 0; i < lambda.length; i++) {
    RealVector x = ed.getEigenvector(i);
    ...
  }
}
```

B.7 Solving Systems of Linear Equations

This section describes standard methods for solving systems of linear equations. Such systems appear widely and frequently in all sorts of engineering problems. Identifying them and knowing about standard solution methods is thus quite important and may save much time in any development process. In addition, the solution techniques presented here are very mature and numerically stable. Note that this section is supposed to give only a brief summary of the topic and practical implementations using the Apache Commons Math library. Further details and the underlying theory can be found in most linear algebra textbooks (e.g., [145, 190]).

Systems of linear equations generally come in the form

$$\begin{pmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} \\ A_{2,0} & A_{2,1} & \cdots & A_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m-1,0} & A_{m-1,1} & \cdots & A_{m-1,n-1} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{m-1} \end{pmatrix}, \quad \text{(B.41)}$$

or, in the standard notation,

$$\mathbf{A} \cdot \boldsymbol{x} = \boldsymbol{b}, \quad \text{(B.42)}$$

where the (known) matrix $\mathbf{A}$ is of size (m, n), the *unknown* vector $\boldsymbol{x}$ is n-dimensional, and the (known) vector $\boldsymbol{b}$ is m-dimensional. Thus n corresponds to the number of unknowns and m to the number

of equations. Each row i of the matrix $\mathbf{A}$ thus represents a single equation

$$A_{i,0}\cdot x_0 + A_{i,1}\cdot x_1 + \ldots + A_{i,n-1}\cdot x_{n-1} = b_i \tag{B.43}$$

$$\text{or} \qquad \sum_{j=0}^{n-1} A_{i,j}\cdot x_j = b_i\,, \tag{B.44}$$

for $i = 0,\ldots,m-1$. Depending on m and n, the following situations may occur:

- If $m = n$ (i.e., $\mathbf{A}$ is square) the number of unknowns matches the number of equations and the system typically (but not always, of course) has a unique solution (see Sec. B.7.1 below).
- If $m < n$, we have more unknowns than equations. In this case no unique solution exists (but possibly infinitely many).
- With $m > n$ the system is said to be *over-determined* and thus not solvable in general. Nevertheless, this is a frequent case that is typically handled by calculating a minimum least squares solution (see Sec. B.7.2).

B.7.1 Exact Solutions

If the number of equations (m) is equal to the number of unknowns (n) and the resulting (square) matrix $\mathbf{A}$ is non-singular and of full rank $m = n$, the system $\mathbf{A}\cdot\boldsymbol{x} = \boldsymbol{b}$ can be expected to have a unique solution for $\boldsymbol{x}$. For example, the system[8]

$$\begin{aligned} 2\cdot x_0 + 3\cdot x_1 - 2\cdot x_2 &= 1,\\ -x_0 + 7\cdot x_1 + 6\cdot x_2 &= -2,\\ 4\cdot x_0 - 3\cdot x_1 - 5\cdot x_2 &= 1, \end{aligned} \tag{B.45}$$

with

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & -2\\ -1 & 7 & 6\\ 4 & -3 & -5 \end{pmatrix}, \qquad \boldsymbol{x} = \begin{pmatrix} x_0\\ x_1\\ x_2 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} 1\\ -2\\ 1 \end{pmatrix}, \tag{B.46}$$

has the unique solution $\boldsymbol{x} = (-0.3698, 0.1780, -0.6027)^\intercal$. The following code segment shows how the previous example is solved using class `LUDecomposition` of the ACM library:

```
import org.apache...linear.DecompositionSolver;
import org.apache...linear.LUDecomposition;

RealMatrix A = MatrixUtils.createRealMatrix(new double[][]
        {{ 2,  3, -2},
         {-1,  7,  6},
         { 4, -3, -5}});
RealVector b = MatrixUtils.createRealVector(new double[]
         {1, -2, 1});
DecompositionSolver solver =
              new LUDecomposition(A).getSolver();
RealVector x = solver.solve(b);
```

An exception is thrown if the matrix `A` is non-square or singular.

[8] Example taken from the *Apache Commons Math User Guide* [4].

B.7.2 Over-Determined System (Least-Squares Solutions)

If a system of linear equations has more equations than unknowns (i.e., $m > n$) it is over-determined and thus has no exact solution. In other words, there is no vector $\boldsymbol{x}$ that satisfies $\mathbf{A} \cdot \boldsymbol{x} = \boldsymbol{b}$ or

$$\mathbf{A} \cdot \boldsymbol{x} - \boldsymbol{b} = \mathbf{0}. \tag{B.47}$$

Instead, *any* $\boldsymbol{x}$ plugged into Eqn. (B.47) yields some non-zero "residual" vector $\boldsymbol{\epsilon}$, such that

$$\mathbf{A} \cdot \boldsymbol{x} - \boldsymbol{b} = \boldsymbol{\epsilon}. \tag{B.48}$$

A "best" solution is commonly found by minimizing the squared norm of this residual, that is, by searching for $\boldsymbol{x}$ such that

$$\|\mathbf{A} \cdot \boldsymbol{x} - \boldsymbol{b}\|^2 = \|\boldsymbol{\epsilon}\|^2 \to \min . \tag{B.49}$$

Several matrix decompositions can be used for calculating the "least-squares solution" of an over-determined system of linear equations. As a simple example, we add a fourth line ($m = 4$) to the system in Eqns. (B.45) and (B.46) to

$$\mathbf{A} = \begin{pmatrix} 2 & 3 & -2 \\ -1 & 7 & 6 \\ 4 & -3 & -5 \\ 2 & -2 & -1 \end{pmatrix}, \qquad \boldsymbol{x} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} 1 \\ -2 \\ 1 \\ 0 \end{pmatrix}, \tag{B.50}$$

without changing the number of unknowns ($n = 3$). The least-squares solution to this over-determined system is (approx.) $\boldsymbol{x} = (-0.2339, 0.1157, -0.4942)^\mathsf{T}$. The following code segment shows the calculation using the `SingularValueDecomposition` class of the ACM library:

```
import org.apache...linear.DecompositionSolver;
import org.apache...linear.SingularValueDecomposition;

RealMatrix A = MatrixUtils.createRealMatrix(new double[][]
        {{ 2,  3, -2},
         {-1,  7,  6},
         { 4, -3, -5},
         { 2, -2, -1}});
RealVector b = MatrixUtils.createRealVector(new double[]
         {1, -2, 1, 0});
DecompositionSolver solver =
          new SingularValueDecomposition(A).getSolver();
RealVector x = solver.solve(b);
```

Alternatively, an instance of `QRDecomposition` could be used for calculating the least-squares solution. If an *exact* solution exists (see Sec. B.7.1), it is the same as the least-squares solution (with zero residual $\boldsymbol{\epsilon} = \mathbf{0}$).

Appendix C

Calculus

This part outlines selected topics from calculus that may serve as a useful supplement to Chapters 6, 16, 17, 24, and 25, in particular.

C.1 Parabolic Fitting

Given a single-variable (1D), discrete function $g\colon \mathbb{Z} \mapsto \mathbb{R}$, it is sometimes useful to locally fit a quadratic (parabolic) function, for example, for precisely locating a maximum or minimum position.

C.1.1 Fitting a Parabolic Function to Three Sample Points

For a quadratic function (second-order polynomial)

$$y = f(x) = a \cdot x^2 + b \cdot x + c \tag{C.1}$$

with parameters a, b, c to pass through a given set of three sample points $\boldsymbol{p}_i = (x_i, y_i)$, $i = 1, 2, 3$, means that the following three equations must be satisfied:

$$\begin{aligned} y_1 &= a \cdot x_1^2 + b \cdot x_1 + c, \\ y_2 &= a \cdot x_2^2 + b \cdot x_2 + c, \\ y_3 &= a \cdot x_3^2 + b \cdot x_3 + c. \end{aligned} \tag{C.2}$$

Written in the standard matrix form $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, or

$$\begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \tag{C.3}$$

the unknown coefficient vector $\boldsymbol{x} = (a, b, c)^\mathsf{T}$ is directly found as

$$\boldsymbol{x} = \boldsymbol{A}^{-1} \cdot \boldsymbol{b} = \begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}, \tag{C.4}$$

assuming that the matrix $\boldsymbol{A}$ has a non-zero determinant. Geometrically this means that the points $\boldsymbol{p}_i$ must not be *collinear*.

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9

Example:

Fitting the sample points $\boldsymbol{p}_1 = (-2,5)^\intercal$, $\boldsymbol{p}_2 = (-1,6)^\intercal$, $\boldsymbol{p}_3 = (3,-10)^\intercal$ to a quadratic function, the equation to solve is (analogous to Eqn. (C.3))

$$\begin{pmatrix} 4 & -2 & 1 \\ 1 & -1 & 1 \\ 9 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 5 \\ 6 \\ -10 \end{pmatrix}, \tag{C.5}$$

with the solution

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} 4 & -2 & 1 \\ 1 & -1 & 1 \\ 9 & 3 & 1 \end{pmatrix}^{-1} \cdot \begin{pmatrix} 5 \\ 6 \\ -10 \end{pmatrix} = \frac{1}{20} \cdot \begin{pmatrix} 4 & -5 & 1 \\ -8 & 5 & 3 \\ -12 & 30 & 2 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 6 \\ -10 \end{pmatrix} = \begin{pmatrix} -1 \\ -2 \\ 5 \end{pmatrix}.$$

Thus $a = -1$, $b = -2$, $c = 5$, and the equation of the quadratic fitting function is $y = -x^2 - 2x + 5$. The result for this example is shown graphically in Fig. C.1.

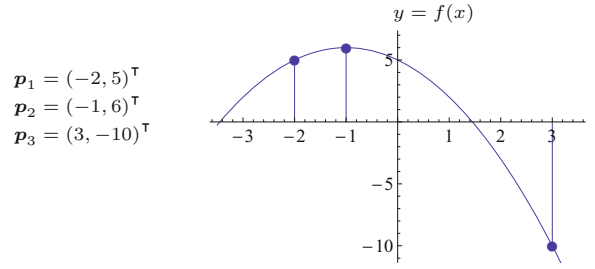

Fig. C.1
Fitting a quadratic function to three arbitrary sample points.

C.1.2 Locating Extrema by Quadratic Interpolation

A special situation is when the given points are positioned at $x_1 = -1$, $x_2 = 0$, and $x_3 = +1$. This is useful, for example, to estimate a continuous extremum position from successive discrete function values defined on a regular lattice. Again the objective is to fit a quadratic function (as in Eqn. (C.1)) to pass through the points $\boldsymbol{p}_1 = (-1,y_1)^\intercal$, $\boldsymbol{p}_2 = (0,y_2)^\intercal$, and $\boldsymbol{p}_3 = (1,y_3)^\intercal$. In this case, the simultaneous equations in Eqn. (C.2) simplify to

$$\begin{aligned} y_1 &= a - b + c, \\ y_2 &= \qquad c, \\ y_3 &= a + b + c, \end{aligned} \tag{C.6}$$

with the solution

$$a = \frac{y_1 - 2 \cdot y_2 + y_3}{2}, \qquad b = \frac{y_3 - y_1}{2}, \qquad c = y_2. \tag{C.7}$$

To estimate a local extremum position, we take the first derivative of the quadratic fitting function (Eqn. (C.1)), which is the linear function $f'(x) = 2a \cdot x + b$, and find the position $\breve{x}$ of its (single) root by solving

$$2a \cdot x + b = 0. \tag{C.8}$$

With a, b taken from Eqn. (C.7), the extremal position is thus found as

$$\breve{x} = \frac{-b}{2a} = \frac{y_1 - y_3}{2 \cdot (y_1 - 2y_2 + y_3)} \ . \tag{C.9}$$

The corresponding extremal *value* can then be found by evaluating the quadratic function $f()$ at position $\breve{x}$, that is,

$$\breve{y} = f(\breve{x}) = a \cdot \breve{x}^2 + b \cdot \breve{x} + c, \tag{C.10}$$

with a, b, c as defined in Eqn. (C.7). Figure C.2 shows an example with sample points $\boldsymbol{p}_1 = (-1, -2)^\mathsf{T}$, $\boldsymbol{p}_2 = (0, 7)^\mathsf{T}$, $\boldsymbol{p}_3 = (1, 6)^\mathsf{T}$. In this case, the interpolated maximum position is at $\breve{x} = 0.4$ and the corresponding maximum value is $f(\breve{x}) = 7.8$.

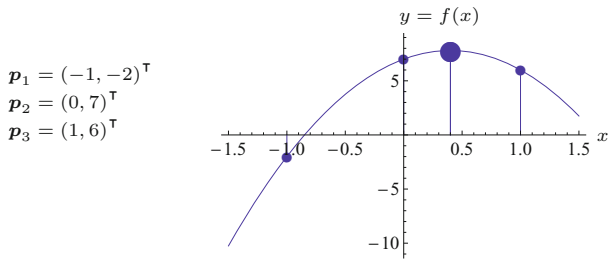

Fig. C.2
Fitting a quadratic function to three reference points at positions $x_1 = -1, x_2 = 0, x_3 = +1$. The interpolated, continuous curve has a maximum at the continuous position $\breve{x} = 0.4$ (large circle).

Using the above scheme, we can interpolate any triplet of successive sample values centered around some position $u \in \mathbb{Z}$, that is, $\boldsymbol{p}_1 = (u-1, y_1)^\mathsf{T}$, $\boldsymbol{p}_2 = (u, y_2)^\mathsf{T}$, $\boldsymbol{p}_3 = (u+1, y_3)^\mathsf{T}$, with arbitrary values y_1, y_2, y_3. In this case the estimated position of the extremum is simply (from Eqn. (C.9))

$$\breve{x} = u + \frac{y_1 - y_3}{2 \cdot (y_1 - 2 \cdot y_2 + y_3)} \ . \tag{C.11}$$

The application of quadratic interpolation to multi-variable functions is described in Sec. C.3.3.

C.2 Scalar and Vector Fields

An RGB color image $\boldsymbol{I}(u, v) = (I_R(u, v), I_G(u, v), I_B(u, v))$ can be considered a 2D function whose values are 3D vectors. Mathematically, this is a special case of a vector-valued function $\boldsymbol{f} \colon \mathbb{R}^n \mapsto \mathbb{R}^m$,

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}(x_0, \ldots, x_{n-1}) = \begin{pmatrix} f_0(\boldsymbol{x}) \\ \vdots \\ f_{m-1}(\boldsymbol{x}) \end{pmatrix}, \tag{C.12}$$

which is composed of m scalar-valued functions $f_i \colon \mathbb{R}^n \mapsto \mathbb{R}$, each being defined on the domain of n-dimensional vectors.

A multi-variable, scalar-valued function $f \colon \mathbb{R}^n \mapsto \mathbb{R}$ is called a *scalar field*, while a vector-valued function $f \colon \mathbb{R}^n \mapsto \mathbb{R}^m$ is referred to as a *vector field*.

C.2.1 The Jacobian Matrix

Assuming that the function $\boldsymbol{f}(\boldsymbol{x}) = (f_0(\boldsymbol{x}), \ldots, f_{m-1}(\boldsymbol{x}))^\mathsf{T}$ is differentiable, the so-called *functional* or *Jacobian* matrix at a specific point $\dot{\boldsymbol{x}} = (\dot{x}_0, \ldots, \dot{x}_{n-1})$ is defined as

$$\mathbf{J}_{\boldsymbol{f}}(\dot{\boldsymbol{x}}) = \begin{pmatrix} \frac{\partial}{\partial x_0} f_0(\dot{\boldsymbol{x}}) & \cdots & \frac{\partial}{\partial x_{n-1}} f_0(\dot{\boldsymbol{x}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_0} f_{m-1}(\dot{\boldsymbol{x}}) & \cdots & \frac{\partial}{\partial x_{n-1}} f_{m-1}(\dot{\boldsymbol{x}}) \end{pmatrix}. \tag{C.13}$$

The Jacobian matrix is of size $m \times n$ and composed of the first derivatives of the m component functions $f_0, \ldots, f_{m-1}$ with respect to each of the n independent variables $x_0, \ldots, x_{n-1}$. Thus each of its elements $\frac{\partial}{\partial x_j} f_i(\dot{\boldsymbol{x}})$ quantifies how much the value of the scalar-valued component function $f_i(\boldsymbol{x}) = f_i(x_0, \ldots, x_{n-1})$ changes when only variable x_j is varied and all other variables remain fixed. Note that the matrix $\mathbf{J}_{\boldsymbol{f}}(\boldsymbol{x})$ is not constant for a given function $\boldsymbol{f}$ but is different at each position $\dot{\boldsymbol{x}}$. In general, the Jacobian matrix is neither square (unless $m = n$) nor symmetric.

C.2.2 Gradients

Gradient of a scalar field

The gradient of a *scalar field* $f\colon \mathbb{R}^n \mapsto \mathbb{R}$, with $f(\boldsymbol{x}) = f(x_0, \ldots, x_{n-1})$, at a given position $\dot{\boldsymbol{x}} \in \mathbb{R}^n$ is defined as

$$(\nabla f)(\dot{\boldsymbol{x}}) = (\operatorname{grad} f)(\dot{\boldsymbol{x}}) = \begin{pmatrix} \frac{\partial}{\partial x_0} f(\dot{\boldsymbol{x}}) \\ \vdots \\ \frac{\partial}{\partial x_{n-1}} f(\dot{\boldsymbol{x}}) \end{pmatrix}. \tag{C.14}$$

The resulting vector-valued function quantifies the amount of output change with respect to changing any of the input variables $x_0, \ldots, x_{n-1}$ at position $\dot{\boldsymbol{x}}$. Thus the gradient of a scalar field is a vector field.

The *directional* gradient of a scalar field describes how the (scalar) function value changes when the coordinates are modified along a particular direction, specified by the unit vector $\mathbf{e}$. We denote the directional gradient as $\nabla_{\mathbf{e}} f$ and define

$$(\nabla_{\mathbf{e}} f)(\dot{\boldsymbol{x}}) = (\nabla f)(\dot{\boldsymbol{x}}) \cdot \mathbf{e}\,, \tag{C.15}$$

where $\cdot$ is the scalar product (see Sec. B.3.1). The result is a scalar value that can be interpreted as the slope of the tangent on the n-dimensional surface of the scalar field at position $\dot{\boldsymbol{x}}$ along the direction specified by the n-dimensional unit vector $\mathbf{e} = (e_0, \ldots, e_{n-1})^\mathsf{T}$.

Gradient of a vector field

To calculate the gradient of a *vector field* $\boldsymbol{f}\colon \mathbb{R}^n \mapsto \mathbb{R}^m$, we note that each row i in the $m \times n$ Jacobian matrix $\mathbf{J}_{\boldsymbol{f}}$ (Eqn. (C.13)) is the transposed gradient vector of the corresponding component function f_i, that is,

$$\mathbf{J}_{\boldsymbol{f}}(\dot{\boldsymbol{x}}) = \begin{pmatrix} (\nabla f_0)(\dot{\boldsymbol{x}})^\intercal \\ \vdots \\ (\nabla f_{m-1})(\dot{\boldsymbol{x}})^\intercal \end{pmatrix}, \tag{C.16}$$

and thus the Jacobian matrix is equivalent to the gradient of the vector field $\boldsymbol{f}$,

$$(\operatorname{grad} \boldsymbol{f})(\dot{\boldsymbol{x}}) \equiv \mathbf{J}_{\boldsymbol{f}}(\dot{\boldsymbol{x}}). \tag{C.17}$$

Analogous to Eqn. (C.15), the *directional* gradient of the vector field is then defined as

$$(\operatorname{grad}_{\mathbf{e}} \boldsymbol{f})(\dot{\boldsymbol{x}}) \equiv \mathbf{J}_{\boldsymbol{f}}(\dot{\boldsymbol{x}}) \cdot \mathbf{e}, \tag{C.18}$$

where $\mathbf{e}$ is again a unit vector specifying the gradient direction and $\cdot$ is the ordinary matrix-vector product. In this case the resulting gradient is a m-dimensional vector with one element for each component function in $\boldsymbol{f}$.

C.2.3 Maximum Gradient Direction

In case of a scalar field $f(\boldsymbol{x})$, a resulting non-zero gradient vector $(\nabla f)(\dot{\boldsymbol{x}})$ (Eqn. (C.14)) is also the direction of the steepest ascent of $f(\boldsymbol{x})$ at position $\dot{\boldsymbol{x}}$.[1] In this case, the L_2 norm (see Sec. B.1.2) of the gradient vector, that is, $\|(\nabla f)(\dot{\boldsymbol{x}})\|$, corresponds to the maximum slope of f at point $\dot{\boldsymbol{x}}$.

In case of a vector field $\boldsymbol{f}(\boldsymbol{x})$, the direction of maximum slope cannot be obtained directly, since the gradient is not a n-dimensional vector but its $m \times n$ Jacobian matrix. In this case, the direction of maximum change in the function $\boldsymbol{f}$ is found as the eigenvector $\boldsymbol{x}_k$ of the square $(n \times n)$ matrix

$$\mathbf{M} = \mathbf{J}_{\boldsymbol{f}}^\intercal(\dot{\boldsymbol{x}}) \cdot \mathbf{J}_{\boldsymbol{f}}(\dot{\boldsymbol{x}}) \tag{C.19}$$

that corresponds to its largest eigenvalue λ_k (see also Sec. B.4).

C.2.4 Divergence of a Vector Field

If the vector field maps to the same vector space (i.e., $\boldsymbol{f} \colon \mathbb{R}^n \mapsto \mathbb{R}^n$), its *divergence* (div) is defined as

$$(\operatorname{div} \boldsymbol{f})(\dot{\boldsymbol{x}}) = \tfrac{\partial}{\partial x_0} f_0(\dot{\boldsymbol{x}}) + \cdots + \tfrac{\partial}{\partial x_{n-1}} f_{n-1}(\dot{\boldsymbol{x}}) \tag{C.20}$$

$$= \sum_{i=0}^{n-1} \tfrac{\partial}{\partial x_i} f_i(\dot{\boldsymbol{x}}) \in \mathbb{R}, \tag{C.21}$$

for a given point $\dot{\boldsymbol{x}}$. The result is a scalar value and thus $(\operatorname{div} \boldsymbol{f})(\dot{\boldsymbol{x}})$ yields a scalar field $\mathbb{R}^n \mapsto \mathbb{R}$. Note that, in this case, the Jacobian matrix $\mathbf{J}_{\boldsymbol{f}}$ in Eqn. (C.13) is square (of size $n \times n$) and $\operatorname{div} \boldsymbol{f}$ is equivalent to the trace of $\mathbf{J}_{\boldsymbol{f}}$, that is,

$$(\operatorname{div} \boldsymbol{f})(\dot{\boldsymbol{x}}) \equiv \operatorname{trace}(\mathbf{J}_{\boldsymbol{f}}(\dot{\boldsymbol{x}})). \tag{C.22}$$

[1] If the gradient vector is *zero*, that is, if $(\nabla f)(\dot{\boldsymbol{x}}) = \mathbf{0}$, the direction of the gradient is undefined at position $\dot{\boldsymbol{x}}$.

C.2.5 Laplacian Operator

The *Laplacian* (or Laplace operator) of a scalar field $f\colon \mathbb{R}^n \mapsto \mathbb{R}$ is a linear differential operator, commonly denoted Δ or ∇^2. The result of applying ∇^2 to the scalar field $f\colon \mathbb{R}^n \mapsto \mathbb{R}$ generates another scalar field that consists of the sum of all unmixed second-order partial derivatives of f (if existent), that is,

$$(\nabla^2 f)(\dot{\boldsymbol{x}}) = \frac{\partial^2}{\partial x_0^2} f(\dot{\boldsymbol{x}}) + \cdots + \frac{\partial^2}{\partial x_{n-1}^2} f(\dot{\boldsymbol{x}}) = \sum_{i=0}^{n-1} \frac{\partial^2}{\partial x_i^2} f(\dot{\boldsymbol{x}}). \tag{C.23}$$

The result is a scalar value that is equivalent to the *divergence* (see Eqn. (C.21)) of the *gradient* (see Eqn. (C.14)) of the scalar field f, that is,

$$(\nabla^2 f)(\dot{\boldsymbol{x}}) = (\mathrm{div}\nabla f)s(\dot{\boldsymbol{x}}). \tag{C.24}$$

The *Laplacian* is also found as the *trace* of the function's Hessian matrix $\mathbf{H}_f$ (see Sec. C.2.6).

For a *vector*-valued function $\boldsymbol{f}\colon \mathbb{R}^n \mapsto \mathbb{R}^m$, the Laplacian at point $\dot{\boldsymbol{x}}$ is again a vector field $\mathbb{R}^n \mapsto \mathbb{R}^m$,

$$(\nabla^2 \boldsymbol{f})(\dot{\boldsymbol{x}}) = \begin{pmatrix} (\nabla^2 f_0)(\dot{\boldsymbol{x}}) \\ (\nabla^2 f_2)(\dot{\boldsymbol{x}}) \\ \vdots \\ (\nabla^2 f_{m-1})(\dot{\boldsymbol{x}}) \end{pmatrix} \in \mathbb{R}^m, \tag{C.25}$$

that is obtained by applying the Laplacian to the individual (scalar-valued) component functions.

C.2.6 The Hessian Matrix

The Hessian matrix of a n-variable, real-valued function $f\colon \mathbb{R}^n \mapsto \mathbb{R}$ is the $n \times n$ square matrix composed of its second-order partial derivatives (assuming they all exist), that is,

$$\mathbf{H}_f = \begin{pmatrix} H_{0,0} & H_{0,1} & \cdots & H_{0,n-1} \\ H_{1,0} & H_{1,1} & \cdots & H_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ H_{n-1,0} & H_{n-1,1} & \cdots & H_{n-1,n-1} \end{pmatrix} \tag{C.26}$$

$$= \begin{pmatrix} \frac{\partial^2}{\partial x_0^2} f & \frac{\partial^2}{\partial x_0 \partial x_1} f & \cdots & \frac{\partial^2}{\partial x_0 \partial x_{n-1}} f \\ \frac{\partial^2}{\partial x_1 \partial x_0} f & \frac{\partial^2}{\partial x_1^2} f & \cdots & \frac{\partial^2}{\partial x_1 \partial x_{n-1}} f \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_{n-1} \partial x_0} f & \frac{\partial^2}{\partial x_{n-1} \partial x_1} f & \cdots & \frac{\partial^2}{\partial x_{n-1}^2} f \end{pmatrix}. \tag{C.27}$$

Since the order of differentiation does not matter (i.e., $H_{i,j} = H_{j,i}$), $\mathbf{H}_f$ is symmetric. Note that the Hessian is a matrix of *functions*. To evaluate the Hessian at a particular point $\dot{\boldsymbol{x}} \in \mathbb{R}^n$, we write

$$\mathbf{H}_f(\dot{\boldsymbol{x}}) = \begin{pmatrix} \frac{\partial^2}{\partial x_0^2} f(\dot{\boldsymbol{x}}) & \cdots & \frac{\partial^2}{\partial x_0 \partial x_{n-1}} f(\dot{\boldsymbol{x}}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_{n-1} \partial x_0} f(\dot{\boldsymbol{x}}) & \cdots & \frac{\partial^2}{\partial x_{n-1}^2} f(\dot{\boldsymbol{x}}) \end{pmatrix}, \tag{C.28}$$

which is a scalar-valued matrix of size $n \times n$. As mentioned already, the *trace* of the Hessian matrix is the *Laplacian* ∇^2 of the function f, that is,

$$\nabla^2 f = \text{trace}\left(\mathbf{H}_f\right) = \sum_{i=0}^{n-1} \frac{\partial^2}{\partial x_i^2} f\,. \tag{C.29}$$

Example

Given a 2D, continuous, grayscale image or scalar-valued intensity function $I(x, y)$, the corresponding Hessian matrix (of size 2×2) contains all second derivatives along the coordinates x, y, that is,

$$\mathbf{H}_I = \begin{pmatrix} \frac{\partial^2}{\partial x^2} I & \frac{\partial^2}{\partial x \partial y} I \\ \frac{\partial^2}{\partial y \partial x} I & \frac{\partial^2}{\partial y^2} I \end{pmatrix} = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{yx} & I_{yy} \end{pmatrix}, \tag{C.30}$$

The elements of $\mathbf{H}_I$ are 2D, scalar-valued functions over x, y and thus scalar fields again. Evaluating the Hessian matrix at a particular point $\dot{\boldsymbol{x}}$ yields the values of the second partial derivatives of I at this position,

$$\mathbf{H}_I(\dot{\boldsymbol{x}}) = \begin{pmatrix} \frac{\partial^2}{\partial x^2} I(\dot{\boldsymbol{x}}) & \frac{\partial^2}{\partial x \partial y} I(\dot{\boldsymbol{x}}) \\ \frac{\partial^2}{\partial y \partial x} I(\dot{\boldsymbol{x}}) & \frac{\partial^2}{\partial y^2} I(\dot{\boldsymbol{x}}) \end{pmatrix} = \begin{pmatrix} I_{xx}(\dot{\boldsymbol{x}}) & I_{xy}(\dot{\boldsymbol{x}}) \\ I_{yx}(\dot{\boldsymbol{x}}) & I_{yy}(\dot{\boldsymbol{x}}) \end{pmatrix}, \tag{C.31}$$

that is, a matrix with scalar-valued elements.

C.3 Operations on Multi-Variable, Scalar Functions (Scalar Fields)

C.3.1 Estimating the Derivatives of a Discrete Function

Images are typically discrete functions (i.e., $I\colon \mathbb{N}^2 \mapsto \mathbb{R}$) and thus not differentiable. The derivatives can nevertheless be estimated by calculating finite differences from the pixel values in a 3×3 neighborhood, which can be expressed as a linear filter or convolution operation ($*$). In particular, the *first*-order derivatives $I_x = \partial I/\partial x$ and $I_y = \partial I/\partial y$ are usually estimated in the form

$$I_x \approx I * \begin{bmatrix} -0.5 & \mathbf{0} & 0.5 \end{bmatrix}, \qquad I_y \approx I * \begin{bmatrix} -0.5 \\ \mathbf{0} \\ 0.5 \end{bmatrix}, \tag{C.32}$$

the second-order derivatives $I_{xx} = \partial^2 I/\partial x^2$ and $I_{yy} = \partial^2 I/\partial y^2$ as

$$I_{xx} \approx I * \begin{bmatrix} 1 & -\mathbf{2} & 1 \end{bmatrix}, \qquad I_{yy} \approx I * \begin{bmatrix} 1 \\ -\mathbf{2} \\ 1 \end{bmatrix}, \tag{C.33}$$

and the mixed derivative

$$\begin{aligned}\frac{\partial^2 I}{\partial x \partial y} &= I_{xy} = I_{yx} \\ &\approx I * \begin{bmatrix}-0.5 & \mathbf{0} & 0.5\end{bmatrix} * \begin{bmatrix}-0.5 \\ \mathbf{0} \\ 0.5\end{bmatrix} = I * \begin{bmatrix}0.25 & 0 & -0.25 \\ 0 & \mathbf{0} & 0 \\ -0.25 & 0 & 0.25\end{bmatrix}. \end{aligned} \tag{C.34}$$

C.3.2 Taylor Series Expansion of Functions

Single-variable functions

The Taylor series expansion (of degree d) of a single-variable function $f\colon \mathbb{R} \mapsto \mathbb{R}$ about a reference point a is

$$\begin{aligned} f(x) &= f(a) + f'(a)\cdot(x-a) + f''(a)\cdot\frac{(x-a)^2}{2} + \cdots \\ &\quad \cdots + f^{(d)}(a)\cdot\frac{(x-a)^d}{d!} + R_d \qquad \text{(C.35)} \\ &= f(a) + \sum_{i=1}^{d} f^{(i)}(a)\cdot\frac{(x-a)^i}{i!} + R_d \qquad \text{(C.36)} \\ &= \sum_{i=0}^{d} f^{(i)}(a)\cdot\frac{(x-a)^i}{i!} + R_d, \qquad \text{(C.37)} \end{aligned}$$

where R_d is the residual term.[2] This means that if the value $f(a)$ and the first d derivatives $f'(a), f''(a), \ldots, f^{(d)}(a)$ exist and are known at some position a, the value of f at *another* point $\dot{x}$ can be estimated (up to the residual R_d) only from the values at point a, without actually evaluating $f(x)$. Omitting the remainder R_d, the result is an *approximation* for $f(\dot{x})$, that is,

$$f(x) \approx \sum_{i=0}^{d} f^{(i)}(a)\cdot\frac{(x-a)^i}{i!}, \tag{C.38}$$

whose accuracy depends upon d and the distance $x - a$.

Multi-variable functions

In general, for a real-valued function of n variables,

$$f(\boldsymbol{x}) = f(x_0, x_2, \ldots, x_{n-1}) \in \mathbb{R},$$

the full Taylor series expansion about a reference point $\boldsymbol{a} = (a_0, \ldots, a_{n-1})^\intercal$ is

$$\begin{aligned} f(x_0, \ldots, x_{n-1}) &= f(\boldsymbol{a}) + \qquad \text{(C.39)} \\ &\sum_{i_0=1}^{\infty}\cdots\sum_{i_{n-1}=1}^{\infty}\Big[\frac{\partial^{i_0}}{\partial x_0^{i_0}}\cdots\frac{\partial^{i_{n-1}}}{\partial x_{n-1}^{i_{n-1}}}\Big] f(\boldsymbol{a})\cdot\frac{(x_0-a_0)^{i_0}\cdots(x_{n-1}-a_{n-1})^{i_{n-1}}}{i_1!\cdots i_n!} \\ = &\sum_{i_1=0}^{\infty}\cdots\sum_{i_n=0}^{\infty}\Big[\frac{\partial^{i_0}}{\partial x_0^{i_0}}\cdots\frac{\partial^{i_{n-1}}}{\partial x_{n-1}^{i_{n-1}}}\Big] f(\boldsymbol{a})\cdot\frac{(x_0-a_0)^{i_0}\cdots(x_{n-1}-a_{n-1})^{i_{n-1}}}{i_0!\cdots i_{n-1}!}. \end{aligned}$$

[2] Note that $f^{(0)} = f$, $f^{(1)} = f'$, $f^{(2)} = f''$ etc., and $1! = 1$.

In Eqn. (C.39),[3] the term

$$\Big[\frac{\partial^{i_0}}{\partial x_0^{i_0}} \cdots \frac{\partial^{i_{n-1}}}{\partial x_{n-1}^{i_{n-1}}}\Big] f(\boldsymbol{a}) \tag{C.40}$$

is the value of the function f, after applying a sequence of n partial derivatives, at the n-dimensional position $\boldsymbol{a}$. The operator $\frac{\partial^i}{\partial x_k^i}$ denotes the i-th partial derivative on the variable x_k.

To formulate Eqn. (C.39) in a more compact fashion, we define the index vector

$$\boldsymbol{i} = (i_0, i_1, \ldots, i_{n-1}), \tag{C.41}$$

(with $i_k \in \mathbb{N}_0$ and thus $\boldsymbol{i} \in \mathbb{N}_0^n$), and the associated operations

$$\begin{aligned} \boldsymbol{i}! &= i_0! \cdot i_1! \cdot \ldots \cdot i_{n-1}!, \\ \boldsymbol{x}^{\boldsymbol{i}} &= x_1^{i_0} \cdot x_2^{i_1} \cdot \ldots \cdot x_{n-1}^{i_{n-1}}, \\ \Sigma \boldsymbol{i} &= i_0 + i_1 + \ldots + i_{n-1}. \end{aligned} \tag{C.42}$$

As a shorthand notation for the combined partial derivative operator in Eqn. (C.40) we define

$$\mathrm{D}^{\boldsymbol{i}} := \frac{\partial^{i_0}}{\partial x_0^{i_0}} \frac{\partial^{i_1}}{\partial x_1^{i_1}} \cdots \frac{\partial^{i_{n-1}}}{\partial x_{n-1}^{i_{n-1}}} = \frac{\partial^{i_0+i_1+\ldots+i_{n-1}}}{\partial x_0^{i_0} \partial x_1^{i_1} \cdots \partial x_{n-1}^{i_{n-1}}}. \tag{C.43}$$

With these definitions, the full Taylor expansion of a multi-variable function about a point $\boldsymbol{a}$, as given in Eqn. (C.39), can be elegantly written in the form

$$f(\boldsymbol{x}) = \sum_{\boldsymbol{i} \in \mathbb{N}_0^n} \mathrm{D}^{\boldsymbol{i}} f(\boldsymbol{a}) \cdot \frac{(\boldsymbol{x}-\boldsymbol{a})^{\boldsymbol{i}}}{\boldsymbol{i}!}. \tag{C.44}$$

Note that $\mathrm{D}^{\boldsymbol{i}} f$ is again a n-dimensional function $\mathbb{R}^n \mapsto \mathbb{R}$, and thus $\big[\mathrm{D}^{\boldsymbol{i}} f\big](\boldsymbol{a})$ in Eqn. (C.44) is the scalar quantity obtained by evaluating the function $[\mathrm{D}^{\boldsymbol{i}} f]$ at the n-dimensional point $\boldsymbol{a}$.

To obtain a Taylor *approximation* of order d, the sum of the indices $i_1, \ldots, i_n$ is limited to d, that is, the summation is constrained to index vectors $\boldsymbol{i}$, with $\Sigma \boldsymbol{i} \leq d$. The resulting formulation,

$$f(\boldsymbol{x}) \approx \sum_{\substack{\boldsymbol{i} \in \mathbb{N}_0^n \\ \Sigma \boldsymbol{i} \leq d}} \mathrm{D}^{\boldsymbol{i}} f(\boldsymbol{a}) \cdot \frac{(\boldsymbol{x}-\boldsymbol{a})^{\boldsymbol{i}}}{\boldsymbol{i}!}, \tag{C.45}$$

is obviously analogous to the 1D case in Eqn. (C.38).

Example: two-variable (2D) function

This example demonstrates the second-order ($d = 2$) Taylor expansion of a 2D ($n = 2$) function $f\colon \mathbb{R}^2 \mapsto \mathbb{R}$ around a point $\boldsymbol{a} = (x_a, y_a)$. By inserting into Eqn. (C.44), we get

[3] Note that symbols $x_0, \ldots, x_{n-1}$ denote the individual variables, while $\dot{x}_0, \ldots, \dot{x}_{n-1}$ are the coordinates of a specific point in n-dimensional space.

$$f(x,y) \approx \sum_{\substack{\boldsymbol{i}\in\mathbb{N}_0^2 \\ \Sigma\boldsymbol{i}\le 2}} \mathrm{D}^{\boldsymbol{i}} f(x_a,y_a)\cdot\frac{1}{\boldsymbol{i}!}\cdot\begin{pmatrix}x-x_a\\ y-y_a\end{pmatrix}^{\boldsymbol{i}} \tag{C.46}$$

$$= \sum_{\substack{0\le i,j\le 2 \\ (i+j)\le 2}} \frac{\partial^{i+j}}{\partial x^i\,\partial y^j} f(x_a,y_a)\cdot\frac{(x-x_a)^i\cdot(y-y_a)^j}{i!\cdot j!}. \tag{C.47}$$

Since $d=2$, the six permissible index vectors $\boldsymbol{i}=(i,j)$, with $\Sigma\boldsymbol{i}\le 2$, are $(0,0)$, $(1,0)$, $(0,1)$, $(1,1)$, $(2,0)$, and $(0,2)$. Inserting into Eqn. (C.47), we obtain the corresponding Taylor approximationat position $(\dot{x},\dot{y})$ as

$$\begin{aligned} f(x,y) \approx{}& \tfrac{\partial^0}{\partial x^0\,\partial y^0} f(x_a,y_a)\cdot\frac{(x-x_a)^0\cdot(y-y_a)^0}{1\cdot 1} \\ &+ \tfrac{\partial^1}{\partial x^1\,\partial y^0} f(x_a,y_a)\cdot\frac{(x-x_a)^1\cdot(y-y_a)^0}{1\cdot 1} \\ &+ \tfrac{\partial^1}{\partial x^0\,\partial y^1} f(x_a,y_a)\cdot\frac{(x-x_a)^0\cdot(y-y_a)^1}{1\cdot 1} \\ &+ \tfrac{\partial^2}{\partial x^1\,\partial y^1} f(x_a,y_a)\cdot\frac{(x-x_a)^1\cdot(y-y_a)^1}{1\cdot 1} \\ &+ \tfrac{\partial^2}{\partial x^2\,\partial y^0} f(x_a,y_a)\cdot\frac{(x-x_a)^2\cdot(y-y_a)^0}{2\cdot 1} \\ &+ \tfrac{\partial^2}{\partial x^0\,\partial y^2} f(x_a,y_a)\cdot\frac{(x-x_a)^0\cdot(y-y_a)^2}{1\cdot 2} \end{aligned} \tag{C.48}$$

$$\begin{aligned} ={}& f(x_a,y_a) \\ &+ \tfrac{\partial}{\partial x} f(x_a,y_a)\cdot(x-x_a) + \tfrac{\partial}{\partial y} f(x_a,y_a)\cdot(y-y_a) \\ &+ \tfrac{\partial^2}{\partial x\,\partial y} f(x_a,y_a)\cdot(x-x_a)\cdot(y-y_a) \\ &+ \frac{1}{2}\cdot\tfrac{\partial^2}{\partial x^2} f(x_a,y_a)\cdot(x-x_a)^2 + \frac{1}{2}\cdot\tfrac{\partial^2}{\partial y^2} f(x_a,y_a)\cdot(y-y_a)^2. \end{aligned} \tag{C.49}$$

It is assumed that the required derivatives of f exist, that is, f is differentiable at point (x_a,y_a) with respect to x and y up to the second order. By slightly rearranging Eqn. (C.49) to

$$\begin{aligned} f(x,y) \approx{}& f(x_a,y_a) + \tfrac{\partial}{\partial x} f(x_a,y_a)\cdot(x-x_a) + \tfrac{\partial}{\partial y} f(x_a,y_a)\cdot(y-y_a) \\ &+ \frac{1}{2}\cdot\Big[\tfrac{\partial^2}{\partial x^2} f(x_a,y_a)\cdot(x-x_a)^2 + 2\cdot\tfrac{\partial^2}{\partial x\,\partial y} f(x_a,y_a)\cdot(x-x_a)\cdot(y-y_a) \\ &\quad + \tfrac{\partial^2}{\partial y^2} f(x_a,y_a)\cdot(y-y_a)^2\Big] \end{aligned} \tag{C.50}$$

we can now write the Taylor expansion in matrix-vector notation as

$$\begin{aligned} f(x,y) \approx \tilde{f}(x,y) = f(x_a,y_a) &+ \Big(\tfrac{\partial}{\partial x} f(x_a,y_a), \tfrac{\partial}{\partial y} f(x_a,y_a)\Big)\cdot\begin{pmatrix}x-x_a\\ y-y_a\end{pmatrix} \\ &+\frac{1}{2}\cdot\Big[(x-x_a,y-y_a)\cdot\begin{pmatrix}\frac{\partial^2}{\partial x^2} f(x_a,y_a) & \frac{\partial^2}{\partial x\,\partial y} f(x_a,y_a)\\ \frac{\partial^2}{\partial x\,\partial y} f(x_a,y_a) & \frac{\partial^2}{\partial y^2} f(x_a,y_a)\end{pmatrix}\cdot\begin{pmatrix}x-x_a\\ y-y_a\end{pmatrix}\Big] \end{aligned} \tag{C.51}$$

or, even more compactly, in the form

$$\tilde{f}(\boldsymbol{x}) = f(\boldsymbol{a}) + \nabla_f^{\intercal}(\boldsymbol{a})\cdot(\boldsymbol{x}-\boldsymbol{a}) + \tfrac{1}{2}\cdot(\boldsymbol{x}-\boldsymbol{a})^{\intercal}\cdot\mathbf{H}_f(\boldsymbol{a})\cdot(\boldsymbol{x}-\boldsymbol{a}). \tag{C.52}$$

Here $\nabla_f^\mathsf{T}(\boldsymbol{a})$ denotes the (transposed) *gradient* vector of the function f at point $\boldsymbol{a}$ (see Sec. C.2.2), and $\mathbf{H}_f$ is the 2×2 *Hessian* matrix of f (see Sec. C.2.6),

$$\mathbf{H}_f(\boldsymbol{a}) = \begin{pmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{pmatrix} = \begin{pmatrix} \frac{\partial^2}{\partial x^2} f(\boldsymbol{a}) & \frac{\partial^2}{\partial x\, \partial y} f(\boldsymbol{a}) \\ \frac{\partial^2}{\partial x\, \partial y} f(\boldsymbol{a}) & \frac{\partial^2}{\partial y^2} f(\boldsymbol{a}) \end{pmatrix}. \tag{C.53}$$

If the function f is *discrete*, for example, a scalar-valued image I, the required partial derivatives at some lattice point $\boldsymbol{a} = (u_a, v_a)^\mathsf{T}$ can be estimated from its 3×3 neighborhood, as described in Sec. C.3.1.

Example: three-variable (3D) function

For a 3D function $f\colon \mathbb{R}^3 \mapsto \mathbb{R}$, the second-order Taylor expansion ($d = 2$) is analogous to Eqns. (C.51–C.52) for the 2D case, except that now the positions $\boldsymbol{x} = (x, y, z)^\mathsf{T}$ and $\boldsymbol{a} = (x_a, y_a, z_a)^\mathsf{T}$ are 3D vectors. The associated (transposed) gradient vector is

$$\nabla_f^\mathsf{T}(\boldsymbol{a}) = \big(\tfrac{\partial}{\partial x} f(\boldsymbol{a}), \tfrac{\partial}{\partial y} f(\boldsymbol{a}), \tfrac{\partial}{\partial z} f(\boldsymbol{a})\big), \tag{C.54}$$

and the Hessian, composed of all second-order partial derivatives, is the 3×3 matrix

$$\mathbf{H}_f(\boldsymbol{a}) = \begin{pmatrix} \frac{\partial^2}{\partial x^2} f(\boldsymbol{a}) & \frac{\partial^2}{\partial x \partial y} f(\boldsymbol{a}) & \frac{\partial^2}{\partial x \partial z} f(\boldsymbol{a}) \\ \frac{\partial^2}{\partial y \partial x} f(\boldsymbol{a}) & \frac{\partial^2}{\partial y^2} f(\boldsymbol{a}) & \frac{\partial^2}{\partial y \partial z} f(\boldsymbol{a}) \\ \frac{\partial^2}{\partial z \partial x} f(\boldsymbol{a}) & \frac{\partial^2}{\partial z \partial y} f(\boldsymbol{a}) & \frac{\partial^2}{\partial z^2} f(\boldsymbol{a}) \end{pmatrix}. \tag{C.55}$$

Note that the order of differentiation is not relevant since, for example, $\frac{\partial^2}{\partial x\, \partial y} = \frac{\partial^2}{\partial y\, \partial x}$, and therefore $\mathbf{H}_f$ is always symmetric.

This can be easily generalized to the n-dimensional case, though things become considerably more involved for Taylor expansions of higher orders ($d > 2$).

C.3.3 Finding the Continuous Extremum of a Multi-Variable Discrete Function

In Sec. C.1.2 we described how the position of a local extremum can be determined by fitting a quadratic function to the neighboring samples of a *1D* function. This section shows how this technique can be extended to n-dimensional, scalar-valued functions $f : \mathbb{R}^n \mapsto \mathbb{R}$.

Without loss of generality we can assume that the Taylor expansion of the function $f(\boldsymbol{x})$ is carried out around the point $\boldsymbol{a} = \mathbf{0} = (0, \ldots, 0)$, which clearly simplifies the remaining formulation. The Taylor approximation function (see Eqn. (C.52)) for this point can be written as

$$\tilde{f}(\boldsymbol{x}) = f(\mathbf{0}) + \nabla_f^\mathsf{T}(\mathbf{0}) \cdot \boldsymbol{x} + \tfrac{1}{2} \cdot \boldsymbol{x}^\mathsf{T} \cdot \mathbf{H}_f(\mathbf{0}) \cdot \boldsymbol{x}, \tag{C.56}$$

with the gradient ∇_f and the Hessian matrix $\mathbf{H}_f$ evaluated at position $\mathbf{0}$. The vector of the first derivative of this function is

$$\tilde{f}'(\boldsymbol{x}) = \nabla_f(\mathbf{0}) + \tfrac{1}{2} \cdot \big[(\boldsymbol{x}^\mathsf{T} \cdot \mathbf{H}_f(\mathbf{0}))^\mathsf{T} + \mathbf{H}_f(\mathbf{0}) \cdot \boldsymbol{x}\big]. \tag{C.57}$$

Since $(\boldsymbol{x}^\mathsf{T} \cdot \mathbf{H}_f)^\mathsf{T} = (\mathbf{H}_f^\mathsf{T} \cdot \boldsymbol{x})$ and because the Hessian matrix $\mathbf{H}_f$ is symmetric (i.e., $\mathbf{H}_f = \mathbf{H}_f^\mathsf{T}$), this simplifies to

$$\begin{aligned} \tilde{f}'(\boldsymbol{x}) &= \nabla_f(\mathbf{0}) + \tfrac{1}{2} \cdot (\mathbf{H}_f(\mathbf{0}) \cdot \boldsymbol{x} + \mathbf{H}_f(\mathbf{0}) \cdot \boldsymbol{x}) && \text{(C.58)} \\ &= \nabla_f(\mathbf{0}) + \mathbf{H}_f(\mathbf{0}) \cdot \boldsymbol{x}. && \text{(C.59)} \end{aligned}$$

A local maximum or minimum is found where all first derivatives $\tilde{f}'$ are zero, so we need to solve

$$\nabla_f(\mathbf{0}) + \mathbf{H}_f(\mathbf{0}) \cdot \breve{\boldsymbol{x}} = \mathbf{0}, \tag{C.60}$$

for the unknown position $\breve{\boldsymbol{x}}$. By multiplying both sides with $\mathbf{H}_f^{-1}$ (assuming that the inverse of $\mathbf{H}_f(\mathbf{0})$ exists), the solution is

$$\breve{\boldsymbol{x}} = -\mathbf{H}_f^{-1}(\mathbf{0}) \cdot \nabla_f(\mathbf{0}), \tag{C.61}$$

for the specific expansion point $\boldsymbol{a} = \mathbf{0}$ (Eqn. (C.63)). Analogously, for an arbitrary expansion point $\boldsymbol{a}$, the extremum position is

$$\breve{\boldsymbol{x}} = \boldsymbol{a} - \mathbf{H}_f^{-1}(\boldsymbol{a}) \cdot \nabla_f(\boldsymbol{a}). \tag{C.62}$$

Note that the inverse Hessian matrix $\mathbf{H}_f^{-1}$ is again symmetric.

The estimated extremal *value* of the approximation function $\tilde{f}$ is found by replacing $\boldsymbol{x}$ in Eqn. (C.56) with the extremal position $\breve{\boldsymbol{x}}$ (calculated in Eqn. (C.61)) as

$$\begin{aligned} \tilde{f}_{\text{extrm}} = \tilde{f}(\breve{\boldsymbol{x}}) &= f(\mathbf{0}) + \nabla_f^\mathsf{T}(\mathbf{0}) \cdot \breve{\boldsymbol{x}} + \tfrac{1}{2} \cdot \breve{\boldsymbol{x}}^\mathsf{T} \cdot \mathbf{H}_f(\mathbf{0}) \cdot \breve{\boldsymbol{x}} \\ &= f(\mathbf{0}) + \nabla_f^\mathsf{T}(\mathbf{0}) \cdot \breve{\boldsymbol{x}} + \tfrac{1}{2} \cdot \breve{\boldsymbol{x}}^\mathsf{T} \cdot \mathbf{H}_f(\mathbf{0}) \cdot (-\mathbf{H}_f^{-1}(\mathbf{0})) \cdot \nabla_f(\mathbf{0}) \\ &= f(\mathbf{0}) + \nabla_f^\mathsf{T}(\mathbf{0}) \cdot \breve{\boldsymbol{x}} - \tfrac{1}{2} \cdot \breve{\boldsymbol{x}}^\mathsf{T} \cdot \mathbf{I} \cdot \nabla_f(\mathbf{0}) && \text{(C.63)} \\ &= f(\mathbf{0}) + \nabla_f^\mathsf{T}(\mathbf{0}) \cdot \breve{\boldsymbol{x}} - \tfrac{1}{2} \cdot \nabla_f^\mathsf{T}(\mathbf{0}) \cdot \breve{\boldsymbol{x}} \\ &= f(\mathbf{0}) + \tfrac{1}{2} \cdot \nabla_f^\mathsf{T}(\mathbf{0}) \cdot \breve{\boldsymbol{x}}, \end{aligned}$$

again for the expansion point $\boldsymbol{a} = \mathbf{0}$.

$$\tilde{f}_{\text{extrm}} = \tilde{f}(\breve{\boldsymbol{x}}) = f(\boldsymbol{a}) + \tfrac{1}{2} \cdot \nabla_f^\mathsf{T}(\boldsymbol{a}) \cdot (\breve{\boldsymbol{x}} - \boldsymbol{a})\,. \tag{C.64}$$

Note that $\tilde{f}_{\text{extrm}}$ may be a local minimum or maximum, but could also be a *saddle point* where the first derivatives of the function are zero as well.

Local extrema in 2D

The aforementioned scheme can be applied to n-dimensional functions. In the special case of a 2D function $f\colon \mathbb{R}^2 \mapsto \mathbb{R}$ (e.g., a 2D image), the gradient vector and the Hessian matrix for the given expansion point $\boldsymbol{a} = (x_a, y_a)^\mathsf{T}$ can be noted as

$$\nabla_f(\boldsymbol{a}) = \begin{pmatrix} d_x \\ d_y \end{pmatrix} \quad \text{and} \quad \mathbf{H}_f(\boldsymbol{a}) = \begin{pmatrix} H_{00} & H_{01} \\ H_{01} & H_{11} \end{pmatrix}, \tag{C.65}$$

for a given expansion point $\boldsymbol{a} = (x_a, y_a)^\mathsf{T}$. In this case, the inverse of the Hessian matrix is

$$\mathbf{H}_f^{-1} = \frac{1}{H_{01}^2 - H_{00} \cdot H_{11}} \cdot \begin{pmatrix} -H_{11} & H_{01} \\ H_{01} & -H_{00} \end{pmatrix} \tag{C.66}$$

and the resulting *position* of the extremal point is (see Eqn. (C.62))

$$\breve{\boldsymbol{x}} = \begin{pmatrix} x_a \\ y_a \end{pmatrix} - \frac{1}{H_{01}^2 - H_{00} \cdot H_{11}} \cdot \begin{pmatrix} -H_{11} & H_{01} \\ H_{01} & -H_{00} \end{pmatrix} \cdot \begin{pmatrix} d_x \\ d_y \end{pmatrix} \tag{C.67}$$

$$= \begin{pmatrix} x_a \\ y_a \end{pmatrix} - \frac{1}{H_{01}^2 - H_{00} \cdot H_{11}} \cdot \begin{pmatrix} H_{01} \cdot d_y - H_{11} \cdot d_x \\ H_{01} \cdot d_x - H_{00} \cdot d_y \end{pmatrix}. \tag{C.68}$$

The extremal position is only defined if the denominator in Eqn. (C.68), $H_{01}^2 - H_{00} \cdot H_{11}$ (equivalent to the determinant of $\mathbf{H}_f$), is non-zero, indicating that the Hessian matrix $\mathbf{H}_f$ is non-singular and thus has an inverse. The associated *value* of $\tilde{f}$ at the estimated extremal position $\breve{\boldsymbol{x}} = (\breve{x}, \breve{y})^\mathsf{T}$ can be now calculated using Eqn. (C.64) as

$$\begin{aligned} \tilde{f}(\breve{x}, \breve{y}) &= f(x_a, y_a) + \tfrac{1}{2} \cdot (d_x, d_y) \cdot \begin{pmatrix} \breve{x} - x_a \\ \breve{y} - y_a \end{pmatrix} \\ &= f(x_a, y_a) + \frac{d_x \cdot (\breve{x} - x_a) + d_y \cdot (\breve{y} - y_a)}{2} \,. \end{aligned} \tag{C.69}$$

Numeric 2D example

The following example shows how a local extremum can be found in a discrete 2D image with sub-pixel accuracy using a second-order Taylor approximation. Assume we are given a grayscale image $I \colon \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{R}$ with the sample values

	u_a-1	u_a	u_a+1
v_a-1	8	11	7
v_a	15	16	9
v_a+1	14	12	10

(C.70)

in the 3×3 neighborhood of position $\boldsymbol{a} = (u_a, v_a)^\mathsf{T}$. Obviously, the discrete center value $f(\boldsymbol{a}) = 16$ is a local maximum but (as we shall see) the maximum of the *continuous* approximation function is *not* at the center. The gradient vector ∇_I and the Hessian Matrix $\mathbf{H}_I$ at the expansion point $\boldsymbol{a}$ are calculated from local finite differences (see Sec. C.3.1) as

$$\nabla_I(\boldsymbol{a}) = \begin{pmatrix} d_x \\ d_y \end{pmatrix} = 0.5 \cdot \begin{pmatrix} 9-15 \\ 12-11 \end{pmatrix} = \begin{pmatrix} -3 \\ 0.5 \end{pmatrix} \qquad \text{and} \tag{C.71}$$

$$\mathbf{H}_I(\boldsymbol{a}) = \begin{pmatrix} H_{11} & H_{12} \\ H_{12} & H_{22} \end{pmatrix} = \begin{pmatrix} 9-2\cdot16+15 & 0.25\cdot(8-14-7+10) \\ 0.25\cdot(8-14-7+10) & 11-2\cdot16+12 \end{pmatrix}$$

$$= \begin{pmatrix} -8.00 & -0.75 \\ -0.75 & -9.00 \end{pmatrix}, \tag{C.72}$$

respectively. The resulting second-order Taylor expansion about the point $\boldsymbol{a}$ is the continuous function (see Eqn. (C.52))

$$\begin{aligned} \tilde{f}(\boldsymbol{x}) &= f(\boldsymbol{a}) + \nabla_I^\mathsf{T}(\boldsymbol{a}) \cdot (\boldsymbol{x} - \boldsymbol{a}) + \tfrac{1}{2} \cdot (\boldsymbol{x} - \boldsymbol{a})^\mathsf{T} \cdot \mathbf{H}_I(\boldsymbol{a}) \cdot (\boldsymbol{x} - \boldsymbol{a}) \\ &= 16 + (-3, 0.5) \cdot \begin{pmatrix} x - u_a \\ y - v_a \end{pmatrix} \\ &\quad + \tfrac{1}{2} \cdot (x - u_a, y - v_a) \cdot \begin{pmatrix} -8.00 & -0.75 \\ -0.75 & -9.00 \end{pmatrix} \cdot \begin{pmatrix} x - u_a \\ y - v_a \end{pmatrix}. \end{aligned} \tag{C.73}$$

We use the inverse of the 2×2 Hessian matrix at position $\boldsymbol{a}$ (see Eqn. (C.66)),

$$\mathbf{H}_I^{-1}(\boldsymbol{a}) = \begin{pmatrix} -8.00 & -0.75 \\ -0.75 & -9.00 \end{pmatrix}^{-1} = \begin{pmatrix} -0.125984 & 0.010499 \\ 0.010499 & -0.111986 \end{pmatrix}, \quad \text{(C.74)}$$

to calculate the *position* of the local extremum $\breve{\boldsymbol{x}}$ (see Eqn. (C.68)) as

$$\begin{aligned} \breve{\boldsymbol{x}} &= \boldsymbol{a} - \mathbf{H}_I^{-1}(\boldsymbol{a}) \cdot \nabla_I(\boldsymbol{a}) \qquad \text{(C.75)} \\ &= \begin{pmatrix} u_a \\ v_a \end{pmatrix} - \begin{pmatrix} -0.125984 & 0.010499 \\ 0.010499 & -0.111986 \end{pmatrix} \cdot \begin{pmatrix} -3 \\ 0.5 \end{pmatrix} = \begin{pmatrix} u_a - 0.3832 \\ v_a + 0.0875 \end{pmatrix}. \end{aligned}$$

Finally, the extremal *value* (see Eqn. (C.64)) is found as

$$\begin{aligned} \tilde{f}(\breve{\boldsymbol{x}}) &= f(\boldsymbol{a}) + \tfrac{1}{2} \cdot \nabla_f^\mathsf{T}(\boldsymbol{a}) \cdot (\breve{\boldsymbol{x}} - \boldsymbol{a}) \\ &= 16 + \tfrac{1}{2} \cdot (-3, 0.5) \cdot \begin{pmatrix} u_a - 0.3832 - u_a \\ v_a + 0.0875 - v_a \end{pmatrix} \qquad \text{(C.76)} \\ &= 16 + \tfrac{1}{2} \cdot (3 \cdot 0.3832 + 0.5 \cdot 0.0875) = 16.5967\,. \end{aligned}$$

Figure (C.3) illustrates the aforementioned example, with the expansion point set to $\boldsymbol{a} = (u_a, v_a)^\mathsf{T} = (0, 0)^\mathsf{T}$.

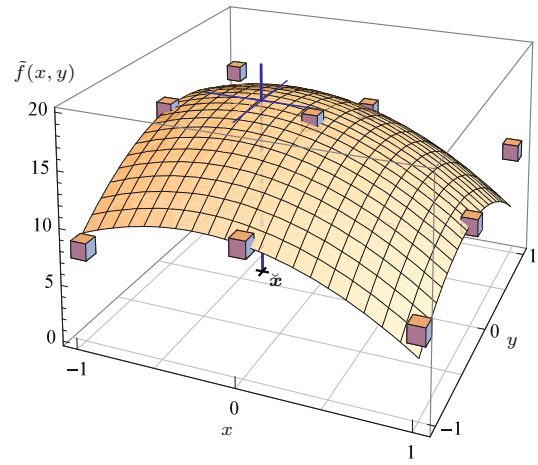

Fig. C.3
Continuous Taylor approximation of a discrete 2D image function for determining the local extremum position with sub-pixel accuracy. The cubes represent the discrete image samples in a 3 × 3 neighborhood around the reference coordinate $(0, 0)$, which is a local maximum of the discrete image function (see Eqn. (C.70) for the concrete values). The parabolic surface shows the continuous approximation $\tilde{f}(x, y)$ obtained by second-order Taylor expansion about the center position $\boldsymbol{a} = (0, 0)$. The vertical line marks the position of the local maximum $\tilde{f}(\breve{\boldsymbol{x}}) = 16.5967$ at $\breve{\boldsymbol{x}} = (-0.3832, 0.0875)$.

Local extrema in 3D

In the case of a three-variable, scalar function $f\colon \mathbb{R}^3 \mapsto \mathbb{R}$, with a given expansion point $\boldsymbol{a} = (x_a, y_a, z_a)^\mathsf{T}$ and

$$\nabla_f(\boldsymbol{a}) = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} \quad \text{and} \quad \mathbf{H}_f(\boldsymbol{a}) = \begin{pmatrix} H_{00} & H_{01} & H_{02} \\ H_{01} & H_{11} & H_{12} \\ H_{02} & H_{12} & H_{22} \end{pmatrix} \qquad \text{(C.77)}$$

being the gradient vector and the Hessian matrix of f at point $\boldsymbol{a}$, respectively, the estimated extremal *position* is

$$\breve{\boldsymbol{x}} = (\breve{x}, \breve{y}, \breve{z})^\intercal = \boldsymbol{a} - \mathbf{H}_f^{-1}(\boldsymbol{a}) \cdot \nabla_f(\boldsymbol{a}) \tag{C.78}$$

$$= \begin{pmatrix} x_a \\ y_a \\ z_a \end{pmatrix} - \frac{1}{H_{02}^2 \cdot H_{11} + H_{01}^2 \cdot H_{22} + H_{00} \cdot H_{12}^2 - H_{00} \cdot H_{11} \cdot H_{22} - 2 \cdot H_{01} \cdot H_{02} \cdot H_{12}}$$

$$\cdot \begin{pmatrix} H_{12}^2 - H_{11} \cdot H_{22} & H_{01} \cdot H_{22} - H_{02} H_{12} & H_{02} \cdot H_{11} - H_{01} \cdot H_{12} \\ H_{01} \cdot H_{22} - H_{02} \cdot H_{12} & H_{02}^2 - H_{00} \cdot H_{22} & H_{00} \cdot H_{12} - H_{01} \cdot H_{02} \\ H_{02} \cdot H_{11} - H_{01} \cdot H_{12} & H_{00} \cdot H_{12} - H_{01} \cdot H_{02} & H_{01}^2 - H_{00} \cdot H_{11} \end{pmatrix} \cdot \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix}.$$

Note that the inverse of the 3×3 Hessian matrix $\mathbf{H}_f^{-1}$ is again symmetric and can be calculated in closed form (as shown in Eqn. (C.78)).[4]

Again using Eqn. (C.64), the estimated extremal *value* at position $\breve{\boldsymbol{x}} = (\breve{x}, \breve{y}, \breve{z})^\intercal$ is found as

$$\tilde{f}(\breve{\boldsymbol{x}}) = f(\boldsymbol{a}) + \tfrac{1}{2} \cdot \nabla_f^\intercal(\boldsymbol{a}) \cdot (\breve{\boldsymbol{x}} - \boldsymbol{a}) \tag{C.79}$$

$$= f(\boldsymbol{a}) + \frac{d_x \cdot (\breve{x} - x_a) + d_y \cdot (\breve{y} - y_a) + d_z \cdot (\breve{z} - z_a)}{2}\,. \tag{C.80}$$

[4] Nevertheless, the use of standard numerical methods is recommended.

Appendix D

Statistical Prerequisites

This part summarizes some essential statistical concepts for vector-valued data, intended as a supplement particularly to Chapters 11 and 17.

D.1 Mean, Variance, and Covariance

For the following definitions we assume a sequence $X = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{n-1})$ of n vector-valued, m-dimensional measurements, with "samples"

$$\boldsymbol{x}_i = (x_{i,0}, x_{i,1}, \ldots, x_{i,m-1})^\mathsf{T} \in \mathbb{R}^m. \tag{D.1}$$

D.1.1 Mean

The n-dimensional *sample mean vector* is defined as

$$\boldsymbol{\mu}(X) = (\mu_0, \mu_1, \ldots, \mu_{m-1})^\mathsf{T} \tag{D.2}$$

$$= \frac{1}{n} \cdot (\boldsymbol{x}_0 + \boldsymbol{x}_1 + \ldots + \boldsymbol{x}_{n-1}) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \boldsymbol{x}_i. \tag{D.3}$$

Geometrically speaking, the vector $\boldsymbol{\mu}(X)$ corresponds to the *centroid* of the sample vectors $\boldsymbol{x}_i$ in m-dimensional space. Each scalar element μ_p is the mean of the associated component (also called *variate* or *dimension*) p over all n samples, that is

$$\mu_p = \frac{1}{n} \cdot \sum_{i=0}^{n-1} x_{i,p}\ , \tag{D.4}$$

for $p = 0, \ldots, m-1$.

D.1.2 Variance and Covariance

The *covariance* quantifies the strength of interaction between a pair of components p, q in the sample X, defined as

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9

$$\sigma_{p,q}(X) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} (x_{i,p} - \mu_p) \cdot (x_{i,q} - \mu_q). \tag{D.5}$$

For efficient calculation, this expression can be rewritten in the form

$$\sigma_{p,q}(X) = \frac{1}{n} \cdot \Big[\underbrace{\sum_{i=0}^{n-1} \left(x_{i,p} \cdot x_{i,q}\right)}_{S_{p,q}(X)} - \frac{1}{n} \cdot \underbrace{\Big(\sum_{i=0}^{n-1} x_{i,p}\Big)}_{S_p(X)} \cdot \underbrace{\Big(\sum_{i=0}^{n-1} x_{i,q}\Big)}_{S_q(X)} \Big], \tag{D.6}$$

which does not require the explicit calculation of μ_p and μ_q. In the special case of $p = q$, we get

$$\sigma_{p,p}(X) = \sigma_p^2(X) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} (x_{i,p} - \mu_p)^2 \tag{D.7}$$

$$= \frac{1}{n} \cdot \Big[\sum_{i=0}^{n-1} x_{i,p}^2 - \frac{1}{n} \cdot \Big(\sum_{i=0}^{n-1} x_{i,p}\Big)^2 \Big], \tag{D.8}$$

which is the *variance within* the component p. This corresponds to the ordinary (one-dimensional) variance $\sigma_p^2(X)$ of the n scalar sample values $x_{0,p}, x_{1,p}, \ldots, x_{n-1,p}$ (see also Sec. 3.7.1).

D.1.3 Biased vs. Unbiased Variance

If the variance (or covariance) of some population is estimated from a small set of random samples, the results obtained by the formulation given in the previous section are known to be statistically *biased*.[1] The most common form of correcting for this bias is to use the factor $1/(n-1)$ instead of $1/n$ in the variance calculations. For example, Eqn. (D.5) would change to

$$\breve{\sigma}_{p,q}(X) = \frac{1}{n-1} \cdot \sum_{i=0}^{n-1} (x_{i,p} - \mu_p) \cdot (x_{i,q} - \mu_q) \tag{D.9}$$

to yield an *unbiased* sample variance. In the following (and throughout the text), we ignore the bias issue and consistently use the factor $1/n$ for all variance calculations. Note, however, that many software packages[2] use the bias-corrected factor $1/(n-1)$ by default and thus may return different results (which can be easily scaled for comparison).

D.2 The Covariance Matrix

The *covariance matrix* Σ for the m-dimensional sample X is a square matrix of size $m \times m$ that is composed of the covariance values $\sigma_{p,q}$ for all pairs (p, q) of components, that is,

[1] Note that the estimation of the mean by the *sample mean* (Eqn. (D.3)) is not affected by this bias problem.

[2] For example, *Apache Commons Math*, *Matlab*, *Mathematica*.

$$\Sigma(X) = \begin{pmatrix} \sigma_{0,0} & \sigma_{0,1} & \cdots & \sigma_{0,m-1} \\ \sigma_{1,0} & \sigma_{1,1} & \cdots & \sigma_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{m-1,0} & \sigma_{m-1,1} & \cdots & \sigma_{m-1,m-1} \end{pmatrix} \tag{D.10}$$

$$= \begin{pmatrix} \sigma_0^2 & \sigma_{0,1} & \cdots & \sigma_{0,m-1} \\ \sigma_{1,0} & \sigma_1^2 & \cdots & \sigma_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{m-1,0} & \sigma_{m-1,1} & \cdots & \sigma_{m-1}^2 \end{pmatrix}. \tag{D.11}$$

Note that any diagonal element of $\Sigma(X)$ is the ordinary (scalar) variance $\sigma_p^2(X)$ (see Eqn. (D.7)), for $p = 0, \ldots, m-1$, which can never be negative. All other entries of a covariance matrix may be positive or negative in general. Since $\sigma_{p,q} = \sigma_{q,p}$, a covariance matrix is always symmetric, with up to $(m^2 + m)/2$ unique elements. Thus, any covariance matrix has the important property of being *positive semidefinite*, which implies that all its eigenvalues (see Sec. B.4) are positive (i.e., non-negative). The covariance matrix can also be written in the form

$$\Sigma(X) = \frac{1}{n} \cdot \sum_{i=0}^{n-1} \underbrace{[\boldsymbol{x}_i - \boldsymbol{\mu}(X)] \cdot [\boldsymbol{x}_i - \boldsymbol{\mu}(X)]^\mathsf{T}}_{= [\boldsymbol{x}_i - \boldsymbol{\mu}(X)] \otimes [\boldsymbol{x}_i - \boldsymbol{\mu}(X)]}, \tag{D.12}$$

where $\otimes$ denotes the outer (vector) product.

The *trace* (sum of the diagonal elements) of the covariance matrix,

$$\sigma_{\text{total}}(X) = \text{trace}\left(\Sigma(X)\right), \tag{D.13}$$

is called the *total variance* of the multivariate sample. Alternatively, the (Frobenius) *norm* of the covariance matrix $\Sigma(X)$, defined as

$$\|\Sigma(X)\|_2 = \Big(\sum_{i=0}^{m-1} \sum_{j=0}^{m-1} \sigma_{i,j}^2\Big)^{1/2}, \tag{D.14}$$

can be used to quantify the overall variance in the sample data.

D.2.1 Example

Assume that the sample X consists of the following set of four 3D vectors (i.e., $m = 3$ and $n = 4$)

$$\boldsymbol{x}_0 = \begin{pmatrix} 75 \\ 37 \\ 12 \end{pmatrix}, \quad \boldsymbol{x}_1 = \begin{pmatrix} 41 \\ 27 \\ 20 \end{pmatrix}, \quad \boldsymbol{x}_2 = \begin{pmatrix} 93 \\ 81 \\ 11 \end{pmatrix}, \quad \boldsymbol{x}_3 = \begin{pmatrix} 12 \\ 48 \\ 52 \end{pmatrix},$$

with each $\boldsymbol{x}_i = (x_{i,\mathrm{R}}, x_{i,\mathrm{G}}, x_{i,\mathrm{B}})^\mathsf{T}$ representing a particular RGB color. The resulting *sample mean vector* (see Eqn. (D.3)) is

$$\boldsymbol{\mu}(X) = \begin{pmatrix} \mu_R \\ \mu_G \\ \mu_B \end{pmatrix} = \frac{1}{4} \cdot \begin{pmatrix} 75 + 41 + 93 + 12 \\ 37 + 27 + 81 + 48 \\ 12 + 20 + 11 + 52 \end{pmatrix} = \frac{1}{4} \cdot \begin{pmatrix} 221 \\ 193 \\ 95 \end{pmatrix} = \begin{pmatrix} 55.25 \\ 48.25 \\ 23.75 \end{pmatrix},$$

and the associated *covariance matrix* (Eqn. (D.11)) is

$$\Sigma(X) = \begin{pmatrix} 972.188 & 331.938 & -470.438 \\ 331.938 & 412.688 & -53.188 \\ -470.438 & -53.188 & 278.188 \end{pmatrix}.$$

As predicted, this matrix is symmetric and all diagonal elements are non-negative. Note that *no* sample bias-correction (see Sec. D.1.3) was used in this example. The *total variance* (Eqn. (D.13)) of the sample set is

$$\sigma_{\text{total}}(X) = \text{trace}\,(\Sigma(X)) = 972.188 + 412.688 + 278.188 \approx 1663.06,$$

and the *Froebenius norm* of the covariance matrix (see Eqn. (D.14)) is $\|\Sigma(X)\|_2 \approx 1364.36$.

D.2.2 Practical Calculation

The calculation of covariance matrices is implemented in almost any software package for statistical analysis or linear algebra. For example, with the *Apache Commons Math* library this could be accomplished as follows:

```
import org.apache.commons.math3.stat.correlation.Covariance;
...
double[][] X;          // X[i] is the i-th sample vector
Covariance cov = new Covariance(X, false);   // no bias correction
RealMatrix S = cov.getCovarianceMatrix();
...
```

D.3 Mahalanobis Distance

The Mahalanobis distance[3] [157] is used to measure distances in multi-dimensional distributions. Unlike the Euclidean distance it takes into account the amount of scatter in the distribution and the correlation between features. In particular, the Mahalanobis distance can be used to measure distances in distributions, where the individual components substantially differ in scale. Depending on their scale, a few components (or even a single component) may dominate the ordinary (Euclidean) distance outcome and the "smaller" components have no influence whatsoever.

D.3.1 Definition

Given a distribution of m-dimensional samples $X = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1})$, with $\boldsymbol{x}_k \in \mathbb{R}^m$, the Mahalanobis distance between two samples $\boldsymbol{x}_a$, $\boldsymbol{x}_b$ is defined as

$$\text{d}_{\text{M}}(\boldsymbol{x}_a, \boldsymbol{x}_b) = \|\boldsymbol{x}_a - \boldsymbol{x}_b\|_{\text{M}} = \sqrt{(\boldsymbol{x}_a - \boldsymbol{x}_b)^\intercal \cdot \boldsymbol{\Sigma}^{-1} \cdot (\boldsymbol{x}_a - \boldsymbol{x}_b)}, \quad \text{(D.15)}$$

where $\boldsymbol{\Sigma}$ is the $m \times m$ *covariance matrix* of the distribution X, as described in Sec. D.2.[4]

[3] http://en.wikipedia.org/wiki/Mahalanobis_distance.

[4] Note that the expression under the root in Eqn. (D.15) is the (dot) product of a row vector and a column vector, that is, the result is a non-negative scalar value.

The Mahalanobis distance normalizes each feature component to *zero mean* and *unit variance*. This makes the distance calculation independent of the scale of the individual components, that is, all components are "treated fairly" even if their range is many orders of magnitude different. In other words, no component can dominate the others even if its magnitude is disproportionally large.

D.3.2 Relation to the Euclidean Distance

Recall that the Euclidean distance between two points $\boldsymbol{x}_a, \boldsymbol{x}_b$ in $\mathbb{R}^m$ is equivalent to the (L2) norm of the difference vector $\boldsymbol{x}_a - \boldsymbol{x}_b$, which can be written in the form

$$\mathrm{d}_{\mathrm{E}}(\boldsymbol{x}_a, \boldsymbol{x}_b) = \|\boldsymbol{x}_a - \boldsymbol{x}_b\|_2 = \sqrt{(\boldsymbol{x}_a - \boldsymbol{x}_b)^{\mathsf{T}} \cdot (\boldsymbol{x}_a - \boldsymbol{x}_b)}. \tag{D.16}$$

Note the structural similarity with the definition of the Mahalanobis distance in Eqn. (D.15), the only difference being the missing matrix $\mathbf{\Sigma}^{-1}$. This becomes even clearer if we analogously insert the identity matrix $\mathbf{I}$ into Eqn. (D.16), that is,

$$\mathrm{d}_{\mathrm{E}}(\boldsymbol{x}_a, \boldsymbol{x}_b) = \|\boldsymbol{x}_a - \boldsymbol{x}_b\|_2 = \sqrt{(\boldsymbol{x}_a - \boldsymbol{x}_b)^{\mathsf{T}} \cdot \mathbf{I} \cdot (\boldsymbol{x}_a - \boldsymbol{x}_b)}, \tag{D.17}$$

which obviously does not change the outcome. The purpose of $\mathbf{\Sigma}^{-1}$ in Eqn. (D.15) is to map the difference vectors (and thus the involved vectors $\boldsymbol{x}_a, \boldsymbol{x}_b$) into a transformed (scaled and rotated) space, where the actual distance measurement is performed. In contrast, with the Euclidean distance, all components contribute equally to the distance measure, without any scaling or other transformation.

D.3.3 Numerical Aspects

For calculating the Mahalobis distance (Eqn. (D.15)) the *inverse* of the covariance matrix (Sec. D.2) is needed. By definition, a covariance matrix $\mathbf{\Sigma}$ is symmetric and its diagonal values are non-negative. Similarly (at least in theory), its inverse $\mathbf{\Sigma}^{-1}$ should also be symmetric with non-negative diagonal values. This is necessary to ensure that the quantities under the square root in Eqn. (D.15) are always positive.

Unfortunately, $\mathbf{\Sigma}$ is often ill-conditioned because of diagonal values that are very small or even zero. In this case, $\mathbf{\Sigma}$ is not positive-definite (as it should be), that is, one or more of its eigenvalues are negative, the inversion becomes numerically unstable and the resulting $\mathbf{\Sigma}^{-1}$ is non-symmetric. A simple remedy to this problem is to add a small quantity to the diagonal of the original covariance matrix $\mathbf{\Sigma}$, that is,

$$\tilde{\mathbf{\Sigma}} = \mathbf{\Sigma} + \epsilon \cdot \mathbf{I}, \tag{D.18}$$

to enforce positive definiteness, and to use $\tilde{\mathbf{\Sigma}}^{-1}$ in Eqn. (D.15).

A possible alternative is to calculate the *Eigen decomposition*[5] of $\mathbf{\Sigma}$ in the form

[5] See http://mathworld.wolfram.com/EigenDecomposition.html and the class `EigenDecomposition` in the *Apache Commons Math* library.

$$\mathbf{\Sigma} = \mathbf{V} \cdot \mathbf{\Lambda} \cdot \mathbf{V}^\mathsf{T} \tag{D.19}$$

where $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{\Sigma}$ (which may be zero or negative). From this we create a modified diagonal matrix $\tilde{\mathbf{\Lambda}}$ by substituting all non-positive eigenvalues with a small positive quantity ϵ, that is,

$$\tilde{\mathbf{\Lambda}}_{i,i} = \min(\mathbf{\Lambda}_{i,i}, \epsilon). \tag{D.20}$$

(typically $\epsilon \approx 10^{-6}$) and finally calculate the modified covariance matrix as

$$\tilde{\mathbf{\Sigma}} = \mathbf{V} \cdot \tilde{\mathbf{\Lambda}} \cdot \mathbf{V}^\mathsf{T}, \tag{D.21}$$

which should be positive definite. The (symmetric) inverse $\tilde{\mathbf{\Sigma}}^{-1}$ is then used in Eqn. (D.15).

D.3.4 Pre-Mapping Data for Efficient Mahalanobis Matching

Assume that we have a large set of sample vectors ("data base") $X = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1})$ which shall be frequently queried for the instance most similar (i.e., closest) to a given search sample $\boldsymbol{x}_\mathrm{s}$. Assuming that the search through X is performed linearly, we would need to calculate $d_\mathrm{M}(\boldsymbol{x}_\mathrm{s}, \boldsymbol{x}_i)$—using Eqn. (D.15)—for all elements of $\boldsymbol{x}_i$ in X.

One way to accelerate the matching is to perform the transformation defined by $\mathbf{\Sigma}^{-1}$ to the entire data set only once, such that the Euclidean norm alone can be used for the distance calculation. For the sake of simplicity we write

$$\mathrm{d}^2_\mathrm{M}(\boldsymbol{x}_a, \boldsymbol{x}_b) = \|\boldsymbol{x}_a - \boldsymbol{x}_b\|^2_\mathrm{M} = \|\boldsymbol{y}\|^2_\mathrm{M} \tag{D.22}$$

with the difference vector $\boldsymbol{y} = \boldsymbol{x}_a - \boldsymbol{x}_b$, such that Eqn. (D.15) becomes

$$\|\boldsymbol{y}\|^2_\mathrm{M} = \boldsymbol{y}^\mathsf{T} \cdot \mathbf{\Sigma}^{-1} \cdot \boldsymbol{y}\,. \tag{D.23}$$

The goal is to find a transformation $\mathbf{U}$ such that we can calculate the Mahalanobis distance from the transformed vectors directly as

$$\hat{\boldsymbol{y}} = \mathbf{U} \cdot \boldsymbol{y}, \tag{D.24}$$

by using the ordinary *Euclidean* norm $\|\cdot\|_2$ instead, that is, in the form

$$\begin{aligned} \|\boldsymbol{y}\|^2_\mathrm{M} &= \|\hat{\boldsymbol{y}}\|^2_2 = \hat{\boldsymbol{y}}^\mathsf{T} \cdot \hat{\boldsymbol{y}} && \text{(D.25)} \\ &= (\mathbf{U} \cdot \boldsymbol{y})^\mathsf{T} \cdot (\mathbf{U} \cdot \boldsymbol{y}) = (\boldsymbol{y}^\mathsf{T} \cdot \mathbf{U}^\mathsf{T}) \cdot (\mathbf{U} \cdot \boldsymbol{y}) && \text{(D.26)} \\ &= \boldsymbol{y}^\mathsf{T} \cdot \mathbf{U}^\mathsf{T} \cdot \mathbf{U} \cdot \boldsymbol{y} = \boldsymbol{y}^\mathsf{T} \cdot \mathbf{\Sigma}^{-1} \cdot \boldsymbol{y}\,. && \text{(D.27)} \end{aligned}$$

While we do not know the matrix $\mathbf{U}$ yet, we see from Eqn. (D.27) that it must satisfy

$$\mathbf{U}^\mathsf{T} \cdot \mathbf{U} = \mathbf{\Sigma}^{-1}. \tag{D.28}$$

Fortunately, since $\mathbf{\Sigma}^{-1}$ is symmetric and positive definite, such a decomposition of $\mathbf{\Sigma}^{-1}$ always exists.

The standard method for calculating $\mathbf{U}$ in Eqn. (D.28) is by the Cholesky decomposition,[6] which can factorize any symmetric, positive definite matrix $\mathbf{A}$ in the form

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\intercal \qquad \text{or} \qquad \mathbf{A} = \mathbf{U}^\intercal \cdot \mathbf{U}, \tag{D.29}$$

where $\mathbf{L}$ is a *lower-triangular* matrix or, alternatively, $\mathbf{U}$ is an *upper-triangular* matrix (the second variant is the one we need).[7] Since the transformation of the difference vectors $\boldsymbol{y} \to \mathbf{U} \cdot \boldsymbol{y}$ is a linear operation, the result is the same if we apply the transformation individually to the original vectors, that is,

$$\hat{\boldsymbol{y}} = \mathbf{U} \cdot \boldsymbol{y} = \mathbf{U} \cdot (\boldsymbol{x}_a - \boldsymbol{x}_b) = \mathbf{U} \cdot \boldsymbol{x}_a - \mathbf{U} \cdot \boldsymbol{x}_b \,. \tag{D.30}$$

This means that, given the transformation $\mathbf{U}$, we can obtain the Mahalanobis distance between two points $\boldsymbol{x}_a, \boldsymbol{x}_b$ (as defined in Eqn. (D.15)) by simply calculating the Euclidean distance in the form

$$\mathrm{d_M}(\boldsymbol{x}_a, \boldsymbol{x}_b) = \|\mathbf{U} \cdot (\boldsymbol{x}_a - \boldsymbol{x}_b)\|_2 = \|\mathbf{U} \cdot \boldsymbol{x}_a - \mathbf{U} \cdot \boldsymbol{x}_b\|_2 \,. \tag{D.31}$$

In summary, this suggests the following solution to a large-database Mahalanobis matching problem:

1. Calculate the covariance matrix $\boldsymbol{\Sigma}$ for the original dataset $X = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1})$.
2. Condition $\boldsymbol{\Sigma}$, such that it is positive definite (see Sec. D.3.3).
3. Find the matrix $\mathbf{U}$, such that $\mathbf{U}^\intercal \cdot \mathbf{U} = \boldsymbol{\Sigma}^{-1}$ (by Cholesky decomposition of $\boldsymbol{\Sigma}^{-1}$).
4. Transform all samples of the original data set $X = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{n-1})$ to $\hat{X} = (\hat{\boldsymbol{x}}_0, \ldots, \hat{\boldsymbol{x}}_{n-1})$, with $\hat{\boldsymbol{x}}_k = \mathbf{U} \cdot \boldsymbol{x}_k$. This now becomes the actual "database".
5. Apply the same transformation to the search sample $\boldsymbol{x}_\mathrm{s}$, that is, calculate $\hat{\boldsymbol{x}}_s = \mathbf{U} \cdot \boldsymbol{x}_\mathrm{s}$.
6. Find the index l of the best-matching element in X (in terms of the Mahalanobis distance) by calculating the *Euclidean* (!) distance between the transformed vectors, that is

$$l = \underset{0 \le k < n}{\operatorname{argmin}} \|\hat{\boldsymbol{x}}_s - \hat{\boldsymbol{x}}_k\|^2 \,. \tag{D.32}$$

Since the matching is now performed with the ordinary Euclidean distance and the Mahalanobis calculation is not required during the search, the savings should be substantial. Also, this opens an easy path to the use of advanced, tree-based matching techniques, such as the common k-nearest neighbor methods.

[6] See http://mathworld.wolfram.com/CholeskyDecomposition.html.

[7] The Cholesky decomposition (CD) requires that the supplied matrix $\mathbf{A}$ is symmetric and positive definite, otherwise the decomposition will fail. In fact, the CD itself is commonly used to test if a given matrix is positive definite. It is implemented by class `CholeskyDecomposition` of the *Apache Commons Math* library.

D.4 The Gaussian Distribution

The Gaussian distribution plays a major role in decision theory, pattern recognition, and statistics in general, because of its convenient analytical properties. A continuous, scalar quantity X is said to be subject to a Gaussian distribution, if the probability of observing a particular value x is

$$p(X=x) = p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x-\mu)^2}{2\cdot\sigma^2}}. \tag{D.33}$$

The Gaussian distribution is completely defined by its mean μ and variance σ^2. The Gaussian distribution, also called a "normal" distribution, is commonly denoted in the form

$$p(x) \sim \mathcal{N}(X\,|\,\mu,\sigma^2) \qquad \text{or} \qquad X \sim \mathcal{N}(\mu,\sigma^2), \tag{D.34}$$

saying that "X is normally distributed with parameters μ and σ^2." As required for any valid probability distribution,

$$\mathcal{N}(X\,|\,\mu,\sigma^2) > 0 \qquad \text{and} \qquad \int_{-\infty}^{\infty} \mathcal{N}(X\,|\,\mu,\sigma^2)\,\mathrm{d}x = 1. \tag{D.35}$$

Thus the area under the probability distribution curve is always one, that is, $\mathcal{N}()$ is normalized. The Gaussian function in Eqn. (D.33) has its maximum height (called "mode") at position $x = \mu$, where its value is

$$p(x=\mu) = \frac{1}{\sqrt{2\pi\sigma^2}}. \tag{D.36}$$

If a random variable X is normally distributed with mean μ and variance σ^2, then the result of a linear mapping of the kind $X' = aX + b$ is again a random variable that is normally distributed, with parameters $\bar{\mu} = a\cdot\mu + b$ and $\bar{\sigma}^2 = a^2\cdot\sigma^2$:

$$X \sim \mathcal{N}(\mu,\sigma^2) \;\Rightarrow\; a\cdot X + b \sim \mathcal{N}(a\cdot\mu+b, a^2\cdot\sigma^2), \tag{D.37}$$

for $a, b \in \mathbb{R}$.

Moreover, if X_1, X_2 are statistically *independent*, normally distributed random variables with means μ_1, μ_2 and variances σ_1^2, σ_2^2, respectively, then a linear combination of the form $a_1X_1 + a_2X_2$ is again normally distributed with $\mu_{12} = a_1\cdot\mu_1 + a_2\cdot\mu_2$ and $\sigma_{12} = a_1^2\cdot\sigma_1^2 + a_2^2\cdot\sigma_2^2$, that is,

$$(a_1X_1 + a_2X_2) \sim \mathcal{N}(a_1\cdot\mu_1 + a_2\cdot\mu_2, a_1^2\cdot\sigma_1^2 + a_2^2\cdot\sigma_2^2). \tag{D.38}$$

D.4.1 Maximum Likelihood Estimation

The probability density function $p(x)$ of a statistical distribution tells us how probable it is to observe the result x for some fixed distribution parameters, such as μ and σ, in case of a normal distribution. If these parameters are *unknown* and need to be estimated,[8] it is interesting to ask the reverse question:

[8] As required, for example, for "minimum error thresholding" in Chapter 11, Sec. 11.1.6.

How likely are particular parameter values for a given set of empirical observations (assuming a certain type of distribution)?

This is (in a casual sense) what the term "likelihood" stands for. In particular, a distribution's *likelihood function* quantifies the probability that a given (fixed) set of observations was generated by some varying distribution parameters.

Note that the probability of observing the outcome x from the normal distribution,

$$p(x) = p(x \,|\, \mu, \sigma^2), \tag{D.39}$$

is really a *conditional* probability, stating how probable it is to observe the value x from a given normal distribution with known parameters μ and σ^2. Conversely, a likelihood function for the normal distribution could be viewed as a conditional function

$$L(\mu, \sigma^2 \,|\, x), \tag{D.40}$$

which quantifies the likelihood of (μ, σ^2) being the correct distribution parameters for a given observation x. The maximum likelihood method tries to find optimal parameters by *maximizing* the value of a distribution's likelihood function L.

If we draw two independent[9] samples x_a, x_b that are subjected to the same distribution, their *joint probability* (i.e., the probability of x_a *and* x_b occurring together in the sample) is the product of their individual probabilities, that is,

$$p(x_a \wedge x_b) = p(x_a) \cdot p(x_b)\,. \tag{D.41}$$

In general, if we are given a vector of m independent observations $X = (x_1, x_2, \ldots, x_m)$ from the same distribution, the probability of observing exactly this set of values is

$$\begin{aligned} p(X) &= p(x_0 \wedge x_1 \wedge \ldots \wedge x_{m-1}) \\ &= p(x_0) \cdot p(x_1) \cdot \ldots \cdot p(x_{m-1}) = \prod_{i=0}^{m-1} p(x_i)\,. \end{aligned} \tag{D.42}$$

Thus, if the sample X originates from a normal distribution $\mathcal{N}$, a suitable likelihood function is

$$L(\mu, \sigma^2 \,|\, X) = p(X \,|\, \mu, \sigma^2) \tag{D.43}$$

$$= \prod_{i=0}^{m-1} \mathcal{N}(x_i \,|\, \mu, \sigma^2) = \prod_{i=0}^{m-1} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{(x_i-\mu)^2}{2\cdot\sigma^2}}. \tag{D.44}$$

The parameters $(\hat{\mu}, \hat{\sigma}^2)$, for which $L(\mu, \sigma^2 \,|\, X)$ is a maximum, are called the maximum-likelihood estimate for X.

Note that it is not necessary for a likelihood function to be a proper (i.e., normalized) probability distribution, since it is only necessary to calculate whether a particular set of distribution parameters

[9] Although this assumption is often violated, independence is important to keep statistical problems simple and tractable. In particular, the values of adjacent image pixels are usually not independent.

is more probable than another. Thus the likelihood function L may be any monotonic function of the corresponding probability p in Eqn. (D.43), in particular its *logarithm*, which is commonly used to avoid multiplying small values.

D.4.2 Gaussian Mixtures

In practice, probabilistic models are often too complex to be described by a single Gaussian (or other standard) distribution. Without losing the mathematical convenience of Gaussian models, highly complex distributions can be modeled as combinations of multiple Gaussian distributions with different parameters. Such a Gaussian *mixture model* is a linear superposition of K Gaussian distributions of the form

$$p(x) = \sum_{j=0}^{K-1} \pi_j \cdot \mathcal{N}(x \,|\, \mu_j, \sigma_j^2), \tag{D.45}$$

where the weights ("mixing coefficients") π_j express the probability that an event x was generated by the j^{th} component (with $\sum_{j=0}^{K-1} \pi_j = 1$).[10] The interpretation of this mixture model is, that there are K independent Gaussian "components" (each with its parameters μ_j, σ_j) that contribute to a common stream of events x_i. If a particular value x is observed, it is assumed to be the result of exactly *one* of the K components, but the identity of that component is unknown.

Assume, as a special case, that a probability distribution $p(x)$ is the superposition (mixture) of *two* Gaussian distributions, that is,

$$p(x) = \pi_a \cdot \mathcal{N}(x \,|\, \mu_a, \sigma_a^2) + \pi_b \cdot \mathcal{N}(x \,|\, \mu_b, \sigma_b^2). \tag{D.46}$$

Any observed value x is assumed to be generated by either the first component (with μ_a, σ_a^2 and prior probability π_a) or the second component (with μ_b, σ_b^2 and prior probability π_b). These parameters as well as the prior probabilities are unknown but can be estimated by maximimizing the likelihood function L. Note that, in general, the unknown parameters cannot be calculated in closed form but only with numerical methods. For further details and solution techniques see [24, 64, 228], for example.

D.4.3 Creating Gaussian Noise

Synthetic Gaussian noise is often used for testing in image processing, particularly for assessing the quality of smoothing filters. While the generation of pseudo-random values that follow a Gaussian distribution is not a trivial task in general,[11] it is readily implemented in Java by the standard class `Random`. For example, the Java method `addGaussianNoise()` in Prog. D.1 adds Gaussian noise with zero mean ($\mu = 0$) and standard deviation `sigma` (σ) to a grayscale image `I` of type `FloatProcessor` (ImageJ). The random values produced

[10] The weight π_j is also called the *prior* probability of the component j.

[11] Typically the so-called *polar method* is used for generating Gaussian random values [138, Sec. 3.4.1].

by successive calls to the method `nextGaussian()` in line 10 follow a Gaussian distribution $\mathcal{N}(0,1)$, with mean $\mu = 0$ and variance $\sigma^2 = 1$. As implied by Eqn. (D.37),

$$X \sim \mathcal{N}(0,1) \quad \Rightarrow \quad a + s \cdot X \sim \mathcal{N}(a, s^2), \tag{D.47}$$

and thus scaling the results from `nextGaussian()` by `s` and additive shifting by a makes the resulting random variable `noise` normally distributed with $\mathcal{N}(a, s^2)$.

```
1  import java.util.Random;
2
3  void addGaussianNoise (FloatProcessor I, double sigma) {
4    int w = I.getWidth();
5    int h = I.getHeight();
6    Random rnd = new Random();
7    for (int v = 0; v < h; v++) {
8      for (int u = 0; u < w; u++) {
9        float val = I.getf(u, v);
10       float noise = (float) (rnd.nextGaussian() * sigma);
11       I.setf(u, v, val + noise);
12     }
13   }
14 }
```

Prog. D.1
Java method for adding Gaussian noise to an image of type `FloatProcessor`.

Appendix E

Gaussian Filters

This part supplements the material presented in Ch. 25 (SIFT).

E.1 Cascading Gaussian Filters

To compute a Gaussian scale space efficiently (as used in the SIFT method, for example), the scale layers are usually not obtained directly from the input image by smoothing with Gaussians of increasing size. Instead, each layer can be calculated recursively from the previous layer by filtering with relatively small Gaussians. Thus, the entire scale space is implemented as a concatenation or "cascade" of smaller Gaussian filters.[1]

If Gaussian filters of sizes σ_1, σ_2 are applied successively to the same image, the resulting smoothing effect is identical to using a single larger Gaussian filter H_σ^G, that is,

$$\left(I * H_{\sigma_1}^{\mathrm{G}}\right) * H_{\sigma_2}^{G} = I * \left(H_{\sigma_1}^{\mathrm{G}} * H_{\sigma_2}^{G}\right) = I * H_{\sigma}^{\mathrm{G}}, \tag{E.1}$$

with $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$ being the size of the resulting combined Gaussian filter H_σ^{G} [129, Sec. 4.5.4]. Put in other words, the *variances* (squares of the σ values) of successive Gaussian filters add up, that is,

$$\sigma^2 = \sigma_1^2 + \sigma_2^2. \tag{E.2}$$

In the special case of the *same* Gaussian filter being applied twice ($\sigma_1 = \sigma_2$), the effective width of the combined filter is $\sigma = \sqrt{2} \cdot \sigma_1$.

E.2 Gaussian Filters and Scale Space

In a Gaussian scale space, the scale corresponding to each level is proportional to the width (σ) of the Gaussian filter required to derive this level from the original (completely unsmoothed) image. Given an image that is already pre-smoothed by a Gaussian filter of width

[1] See Chapter 25, Sec. 25.1.1 for details.

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9

σ_1 and should be smoothed to some target scale $\sigma_2 > \sigma_1$, the required width of the additional Gaussian filter is

$$\sigma_d = \sqrt{\sigma_2^2 - \sigma_1^2}. \tag{E.3}$$

Usually the neighboring layers of the scale space differ by a constant scale factor (κ) and the transformation from one scale level to another can be accomplished by successively applying Gaussian filters. Despite the constant scale factor, however, the width of the required filters is *not* constant but depends on the image's initial scale. In particular, if we want to transform an image with scale σ_0 by a factor κ to a new scale $\kappa \cdot \sigma_0$, then (from Eqn. (E.2)) for σ_d the relation

$$(\kappa \cdot \sigma_0)^2 = \sigma_0^2 + \sigma_d^2 \tag{E.4}$$

must hold. Thus, the width σ_d of the required Gaussian smoothing filter is

$$\sigma_d = \sigma_0 \cdot \sqrt{\kappa^2 - 1}. \tag{E.5}$$

For example, doubling the scale ($\kappa = 2$) of an image that is presmoothed with σ_0 requires a Gaussian filter of width $\sigma_d = \sigma_0 \cdot (2^2 - 1)^{1/2} = \sigma_0 \cdot \sqrt{3} \approx \sigma_0 \cdot 1.732$.

E.3 Effects of Gaussian Filtering in the Frequency Domain

For the 1D Gaussian function

$$g_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{x^2}{2\sigma^2}} \tag{E.6}$$

the continuous Fourier transform[2] $\mathcal{F}(g_\sigma)$ is

$$G_\sigma(\omega) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{\omega^2\sigma^2}{2}}. \tag{E.7}$$

Doubling the width (σ) of a Gaussian filter corresponds to cutting the bandwidth by half. If σ is doubled, the Fourier transform becomes

$$G_{2\sigma}(\omega) = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{\omega^2(2\sigma)^2}{2}} = \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{4\omega^2\sigma^2}{2}} \tag{E.8}$$

$$= \frac{1}{\sqrt{2\pi}} \cdot e^{-\frac{(2\omega)^2\sigma^2}{2}} = G_\sigma(2\omega) \tag{E.9}$$

and, in general, when scaling the filter by a factor k,

$$G_{k\sigma}(\omega) = G_\sigma(k\omega). \tag{E.10}$$

That is, if σ is *increased* (or the kernel widened) by a factor k, the corresponding Fourier transform gets *contracted* by the same factor. In terms of linear filtering this means that widening the kernel by some factor k decimates the resulting signal bandwidth by $\frac{1}{k}$.

[2] See also Chapter 18, Sec. 18.1.

E.4 LoG-Approximation by the DoG

The 2D LoG kernel (see Ch. 25, Sec. 25.1.1),

$$L_\sigma(x,y) = \left(\nabla^2 g_\sigma\right)(x,y) = \frac{1}{\pi\sigma^4}\Big(\frac{x^2+y^2-2\sigma^2}{2\sigma^2}\Big)\cdot e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{E.11}$$

has a (negative) peak at the origin with the associated function value

$$L_\sigma(0,0) = -\frac{1}{\pi\sigma^4}\,. \tag{E.12}$$

Thus, the *scale normalized* LoG kernel, defined in Eqn. (25.10) as

$$\hat{L}_\sigma(x,y) = \sigma^2 \cdot L_\sigma(x,y), \tag{E.13}$$

has the peak value

$$\hat{L}_\sigma(0,0) = -\frac{1}{\pi\sigma^2} \tag{E.14}$$

at the origin. In comparison, for a given scale factor κ, the unscaled DoG function

$$\begin{aligned}\mathrm{DoG}_{\sigma,\kappa}(x,y) &= G_{\kappa\sigma}(x,y) - G_\sigma(x,y)\\ &= \frac{1}{2\pi\kappa^2\sigma^2}\cdot e^{-\frac{x^2+y^2}{2\kappa^2\sigma^2}} - \frac{1}{2\pi\sigma^2}\cdot e^{-\frac{x^2+y^2}{2\sigma^2}},\end{aligned} \tag{E.15}$$

has a peak value

$$\mathrm{DoG}_{\sigma,\kappa}(0,0) = -\frac{\kappa^2-1}{2\pi\kappa^2\sigma^2}\,. \tag{E.16}$$

By scaling the DoG function by some factor λ to match the LoG's center peak value, such that $L_\sigma(0,0) = \lambda \cdot \mathrm{DoG}_{\sigma,\kappa}(0,0)$, the original LoG (Eqn. (E.11)) is approximated by the DoG in the form

$$L_\sigma(x,y) \approx \frac{2\kappa^2}{\sigma^2(\kappa^2-1)} \cdot \mathrm{DoG}_{\sigma,\kappa}(x,y). \tag{E.17}$$

Similarly, the scale-normalized LoG (Eqn. (E.13)) is approximated by the DoG as[3]

$$\hat{L}_\sigma(x,y) \approx \frac{2\kappa^2}{\kappa^2-1} \cdot \mathrm{DoG}_{\sigma,\kappa}(x,y). \tag{E.18}$$

Since the factor in Eqn. (E.18) depends on κ only, the DoG approximation is (for a constant size ratio κ) implicitly proportional to the scale normalized LoG for any scale σ.

[3] A different formulation, $\hat{L}_\sigma(x,y) \approx \frac{1}{\kappa-1} \cdot \mathrm{DoG}_{\sigma,\kappa}(x,y)$, is given in [153], which is the same as Eqn. (E.18) for $\kappa \to 1$, but not for $\kappa > 1$. The essence is that the leading factor is constant and independent of σ, and can thus be ignored when comparing the magnitude of the filter responses at varying scales.

Appendix F

Java Notes

As a text for undergraduate engineering curricula, this book assumes basic programming skills in a procedural language, such as Java, C#, or C. The examples in the main text should be easy to understand with the help of an introductory book on Java or one of the many online tutorials. Experience shows, however, that difficulties with some basic Java concepts pertain and often cause complications, even at higher levels. The following sections address some of these typical problem spots.

F.1 Arithmetic

Java is a "strongly typed" programming language, which means in particular that any variable has a fixed type that cannot be altered dynamically. Also, the result of an expression is determined by the types of the involved operands and *not* (in the case of an assignment) by the type of the "receiving" variable.

F.1.1 Integer Division

Division involving integer operands is a frequent cause of errors. If the variables `a` and `b` are both of type `int`, then the expression `a / b` is evaluated according to the rules of integer division. The result—the number of times `b` is contained in `a`—is again of type `int`. For example, after the Java statements

```
int a = 2;
int b = 5;
double c = a / b;    // resulting value of c is zero!
```

the value of `c` is *not* 0.4 but 0.0 because the expression `a / b` on the right yields the `int`-value 0, which is then automatically converted to the `double` value 0.0.

If we wanted to evaluate `a / b` as a *floating-point* operation (as most pocket calculators do), at least one of the involved operands

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9

must be converted to a floating-point value, such as by an explicit type cast, for example,

```
double c = (double) a / b;      // value of c is 0.4
```

or alternatively

```
double c = a / (double) b;      // value of c is 0.4
```

Example

Assume, for example, that we want to scale any pixel value a of an image such that the maximum pixel value $a_{\max}$ is mapped to 255 (see Ch. 4). In mathematical notation, the scaling of the pixel values is simply expressed as

$$c \leftarrow \frac{a_i}{a_{\max}} \cdot 255$$

and it may be tempting to convert this 1:1 into Java code, such as

```
int a_max = ip.getMaxValue();
for ... {
  int a = ip.getPixel(u,v);
  int c = (a / a_max) * 255;   // ← problem!
  ip.putPixel(u, v, c);
}
...
```

As we can easily predict, the resulting image will be all black (zero values), except those pixels whose value was `a_max` originally (they are set to 255). The reason is again that the division `a / a_max` has two operands of type `int`, and the result is thus zero whenever the denumerator (`a_max`) is greater than the numerator (`a`).

Of course, the entire operation could be performed in the floating-point domain by converting one of the operands (as we have shown), but this is not even necessary in this case. Instead, we may simply swap the order of operations and start with the multiplication:

```
int c = a * 255 / a_max;
```

Why does this work now? The subexpression `a * 255` is evaluated first,[1] generating large intermediate values that pose no problem for the subsequent (integer) division. Nevertheless, *rounding* should always be considered to obtain more accurate results when computing fractions of integers (see Sec. F.1.5).

F.1.2 Modulus Operator

The result of the modulus operator $a \bmod b$ (used in several places in the main text) is defined [92, p. 82] as the remainder of the "floored" division a/b,

$$a \bmod b \equiv \begin{cases} a & \text{for } b = 0, \\ a - b \cdot \lfloor a/b \rfloor & \text{otherwise,} \end{cases} \tag{F.1}$$

[1] In Java, expressions at the same level are always evaluated in left-to-right order, and therefore no parentheses are required in this example (though they would do no harm either).

for $a, b \in \mathbb{R}$. This type of operator or library method was not available in the standard Java API until recently.[2] The following Java method implements the mod operation according to the definition in Eqn. (F.1):[3]

```
int Mod(int a, int b) {
  if (b == 0)
    return a;
  if (a * b >= 0)
    return a - b * (a / b);
  else
    return a - b * (a / b - 1);
}
```

Note that the *remainder* operator `%`, defined as

$$a \,\%\, b \;\equiv\; a - b \cdot \text{truncate}(a/b), \qquad \text{for } b \neq 0, \tag{F.2}$$

is often used in this context, but yields the same results only for *positive* operands $a \geq 0$ and $b > 0$. For example,

$$\begin{array}{rcrcr} 13 & \text{mod} & 4 & = & 1 \\ 13 & \text{mod} & -4 & = & -3 \\ -13 & \text{mod} & 4 & = & 3 \\ -13 & \text{mod} & -4 & = & -1 \end{array} \quad \text{vs.} \quad \begin{array}{rcrcr} 13 & \% & 4 & = & 1 \\ 13 & \% & -4 & = & 1 \\ -13 & \% & 4 & = & -1 \\ -13 & \% & -4 & = & -1 \end{array}$$

F.1.3 Unsigned Byte Data

Most grayscale and indexed images in Java and ImageJ are composed of pixels of type `byte`, and the same holds for the individual components of most color images. A single byte consists of eight bits and can thus represent $2^8 = 256$ different bit patterns or values, usually mapped to the numeric range $0, \ldots, 255$. Unfortunately, Java (unlike C and C++) does *not* provide a suitable "unsigned" 8-bit data type. The primitive Java type `byte` is "signed", using one of its eight bits for the $\pm$ sign, and is intended to hold values in the range $-128, \ldots, +127$.

Java's `byte` data can still be used to represent the values 0 to 255, but conversions must take place to perform proper arithmetic computations. For example, after execution of the statements

```
int  a = 200;
byte b = (byte) p;
```

the variables `a` (32-bit `int`) and `b` (8-bit `byte`) contain the binary patterns

```
a = 00000000000000000000000011001000
b =                         11001000
```

Interpreted as a (signed) `byte` value, with the leftmost bit[4] as the sign bit, the variable `b` has the decimal value -56. Thus after the statement

[2] Starting with Java version 1.8 the mod operation (as defined in Eqn. (F.1)) is implemented by the standard method `Math.floorMod(a, b)`.

[3] The definition in Eqn. (F.1) is not restricted to integer operands.

[4] Java uses the standard "2s-complement" representation, where a sign bit = 1 stands for a negative value.

```
int  a1 = b;                // a1 == -56
```

the value of the new `int` variable `a1` is −56! To (ab-)use signed `byte` data as *unsigned* data, we can circumvent Java's standard conversion mechanism by disguising the content of `b` as a logic (i.e., nonarithmetic) *bit pattern*; for example, by

```
int  a2 = (0xff & b);       // a2 == 200
```

where `0xff` (in hexadecimal notation) is an `int` value with the binary bit pattern `00000000000000000000000011111111` and `&` is the bitwise AND operator. Now the variable `a2` contains the right integer value (200) and we thus have a way to use Java's (signed) `byte` data type for storing *unsigned* values. Within ImageJ, access to pixel data is routinely implemented in this way, which is considerably faster than using the convenience methods `getPixel()` and `putPixel()`.

F.1.4 Mathematical Functions in Class `Math`

Java provides most standard mathematical functions as static methods in class `Math`, as listed in Table F.1. The `Math` class is part of the `java.lang` package and thus requires no explicit import to be used. Most `Math` methods accept arguments of type `double` and also return values of type `double`. As a simple example, a typical use of the cosine function $y = \cos(x)$ is

```
double x;
double y = Math.cos(x);
```

Similarly, the `Math` class defines some common numerical constants as static variables; for example, the value of π could be obtained by

```
double pi = Math.PI;
```

Table F.1
Mathematical methods and constants defined by Java's `Math` class.

`double abs(double a)`	`double max(double a, double b)`
`int abs(int a)`	`float max(float a, float b)`
`float abs(float a)`	`int max(int a, int b)`
`long abs(long a)`	`long max(long a, long b)`
`double ceil(double a)`	`double min(double a, double b)`
`double floor(double a)`	`float min(float a, float b)`
`int floorMod(int a, int b)`	`int min(int a, int b)`
`long floorMod(long a, long b)`	`long min(long a, long b)`
`double rint(double a)`	
`long round(double a)`	`double random()`
`int round(float a)`	
`double toDegrees(double rad)`	`double toRadians(double deg)`
`double sin(double a)`	`double asin(double a)`
`double cos(double a)`	`double acos(double a)`
`double tan(double a)`	`double atan(double a)`
`double atan2(double y, double x)`	
`double log(double a)`	`double exp(double a)`
`double sqrt(double a)`	`double pow(double a, double b)`
`double E`	`double PI`

F.1.5 Numerical Rounding

Java's `Math` class (confusingly) offers three different methods for rounding floating-point values:

```
double rint(double x)
long   round(double x)
int    round(float x)
```

For example, a `double` value `x` can be rounded to `int` in any of the following ways:

```
double x; int k;
k = (int) Math.rint(x);
k = (int) Math.round(x);
k = Math.round((float) x);
```

If the operand `x` is known to be positive (as is typically the case with pixel values) rounding can be accomplished without using any method calls by

```
k = (int) (x + 0.5);    // only if x >= 0
```

In this case, the expression `(x + 0.5)` is first computed as a floating-point (`double`) value, which is then truncated (toward zero) by the explicit `(int)` typecast.

F.1.6 Inverse Tangent Function

The inverse tangent function $\varphi = \tan^{-1}(a)$ or $\varphi = \arctan(a)$ is used in several places in the main text. This function is implemented by the method `atan(double a)` in Java's `Math` class (Table F.1). The return value of `atan()` is in the range $[-\frac{\pi}{2}, \ldots, \frac{\pi}{2}]$ and thus restricted to only two of the four quadrants. Without any additional constraints, the resulting angle is ambiguous. In many practical situations, however, a is given as the ratio of two catheti $(\Delta x, \Delta y)$ of a right-angled triangle in the form

$$\varphi = \arctan\left(\frac{y}{x}\right), \tag{F.3}$$

for which we introduced the two-parameter function

$$\varphi = \mathrm{ArcTan}(x, y) \tag{F.4}$$

in the main text. The function $\mathrm{ArcTan}(x, y)$ is implemented by the standard method `atan2(dy,dx)` in Java's `Math` class (note the reversed parameters though) and returns an unambiguous angle φ in the range $[-\pi, \ldots, \pi]$; that is, in any of the four quadrants of the unit circle.[5] Also, the `atan2()` method returns a useful value even if both arguments are zero.

[5] The function `atan2(dy,dx)` is available in most current programming languages, including Java, C, and C++.

F.1.7 Classes `Float` and `Double`

The representation of floating-point numbers in Java follows the IEEE standard, and thus the types `float` and `double` include the values

`Float.MIN_VALUE`,	`Double.MIN_VALUE`,
`Float.MAX_VALUE`,	`Double.MAX_VALUE`,
`Float.POSITIVE_INFINITY`,	`Double.POSITIVE_INFINITY`,
`Float.NEGATIVE_INFINITY`,	`Double.NEGATIVE_INFINITY`,
`Float.NaN`,	`Double.NaN`.

These values are defined as constants in the corresponding wrapper classes `Float` and `Double`, respectively. If any `INFINITY` or `NaN`[6] value occurs in the course of a computation (e.g., as the result of dividing by zero),[7] Java continues without raising an error, so incorrect values may ripple through a whole chain of calculations, making the actual bugs difficult to locate.

F.1.8 Testing Floating-Point Values Against Zero

Comparing floating-point values or testing them for zero is a non-trivial issue and a frequent cause of errors. In particular, one should *never* write

```
if (x == 0.0) {...}   ← problem!
```

if `x` is a floating-point variable. This is often needed, for example, to make sure that it is safe to divide another quantity by `x`. The aforementioned test, however, is not sufficient since `x` may be non-zero but still too small as a divisor.

A much better alternative is to test if `x` is "close" to zero, that is, within some small positive/negative (*epsilon*) interval. While the proper choice of this interval depends on the specific situation, the following settings are usually sufficient for safe operation:[8]

```
static final float  EPSILON_FLOAT  = 1e-7f;
static final double EPSILON_DOUBLE = 2e-16;

float x;
double y;

if (Math.abs(x) < EPSILON_FLOAT) {
  ... // x is practically zero
}

if (Math.abs(y) < EPSILON_DOUBLE) {
  ... // y is practically zero
}
```

[6] `NaN` stands for "not a number".

[7] In Java, this only holds for floating-point operations, whereas integer division by zero always causes an *exception*.

[8] These settings account for the limited *machine accuracy* (ϵ_m) of the IEEE 754 standard types `float` ($\epsilon_m \approx 1.19 \cdot 10^{-7}$) and `double` ($\epsilon_m \approx 2.22 \cdot 10^{-16}$) [190, Ch. 1, Sec. 1.1.2].

F.2 Arrays in Java

F.2.1 Creating Arrays

Unlike in most traditional programming languages (such as FORTRAN or C), arrays in Java can be created *dynamically*, meaning that the size of an array can be specified at runtime using the value of some variable or arithmetic expression. For example:

```
int N = 20;
int[] A = new int[N];
int[] B = new int[N * N];
```

Once allocated, however, the size of any Java array is fixed and cannot be subsequently altered.[9] Note that Java arrays may be of length *zero*!

After its definition, an array variable can be assigned any other compatible array or the constant value `null`, for example, [10]

```
A = B;     // A now references the data in B
B = null;
```

With the assignment `A = B`, the array initially referenced by `A` becomes unaccessible and thus turns into *garbage*. In contrast to C and C++, where unnecessary storage needs to be *de*allocated explicitly, this is taken care of in Java by its built-in "garbage collector". It is also convenient that newly created arrays of numerical element types (`int`, `float`, `double`, etc.) are automatically initialized to zero.

F.2.2 Array Size

Since an array may be created dynamically, it is important that its actual size can be determined at runtime. This is done by accessing the `length` attribute[11]

```
int k = A.length;  // number of elements in A
```

The size is a property of the array itself and can therefore be obtained inside any method from array arguments passed to it. Thus (unlike in C, for example) it is not necessary to pass the size of an array as a separate function argument.

If an array has more than one dimension, the size (`length`) along every dimension must be queried separately (see Sec. F.2.4). Also arrays are not necessarily rectangular; for example, the rows of a 2D array may have different lengths (including zero).

F.2.3 Accessing Array Elements

In Java, the index of the first array element is always 0 and the index of the last element is $N-1$ for an array with a total of N elements. To iterate through a 1D array `A` of arbitrary size, one would typically use a construct like

[9] For additional flexibility, Java provides a number of universal container classes (e.g., the classes `Set` and `List`) for a wide range of applications.

[10] This is not possible if the array variable was defined with the `final` attribute.

[11] Notice that the `length` attribute of an array is not a method!

```
for (int i = 0; i < A.length; i++) {
    // do something with A[i]
}
```

Alternatively, if only the array *values* are relevant and the array *index* (`i`) is not needed, one could use to following (even simpler) loop construct:

```
for (int a : A) {
    // do something with array values a
}
```

In both cases, the Java compiler can generate very efficient runtime code, since the source code makes obvious that the `for` loop does not access any elements outside the array limits and thus no explicit boundary checking is needed at execution time. This fact is very important for implementing efficient image processing programs in Java.

Images in Java and ImageJ are usually stored as 1D arrays (accessible through the `ImageProcessor` method `getPixels()` in ImageJ), with pixels arranged in row-first order.[12] Statistical calculations and most point operations can thus be efficiently implemented by directly accessing the underlying 1D array. For example, the `run` method of the contrast enhancement plugin in Prog. 4.1 (see Chapter 4, p. 58) could also be implemented in the following manner:

```
public void run(ImageProcessor ip) {
  // ip is assumed to be of type ByteProcessor
  byte[] pixels = (byte[]) ip.getPixels();
  for (int i = 0; i < pixels.length; i++) {
    int a = 0xFF & pixels[i];            // direct read operation
    int b = (int) (a * 1.5 + 0.5);
    if (b > 255)
      b = 255;
    pixels[i] = (byte) (0xFF & b);       // direct write operation
  }
}
```

F.2.4 2D Arrays

Multidimensional arrays are a frequent source of confusion. In Java, all arrays are 1D in principle, and multi-dimensional arrays are implemented as 1D arrays of arrays etc. (see Fig. F.1). If, for example, the 3×3 matrix

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \tag{F.5}$$

is defined as a 2D `int` array,

```
int[][] A = {{1,2,3},
             {4,5,6},
             {7,8,9}};
```

[12] This means that horizontally adjacent image pixels are stored next to each other in computer memory.

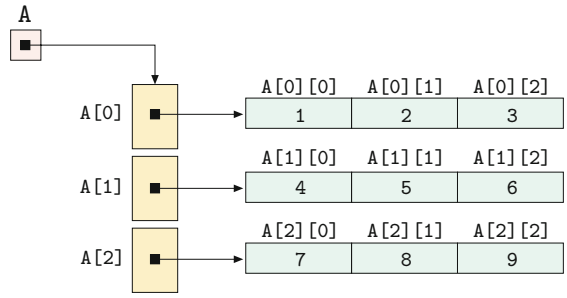

Fig. F.1
Layout of elements of a 2D Java array (corresponding to Eqn. (F.5)). In Java, multidimensional arrays are generally implemented as *1D* arrays whose elements are again 1D arrays.

then `A` is actually a *1D* array with three elements, each of which is again a 1D array. The elements `A[0]`, `A[1]` and `A[2]` are of type `int[]` and correspond to the three rows of the matrix A (see Fig. F.1).

The usual assumption is that the array elements are arranged in *row-first* order, as illustrated in Fig. F.1. The first index thus corresponds to the *row* number r and the second index corresponds to the *column* number c, that is,

$$a_{r,c} \equiv \texttt{A[}r\texttt{][}c\texttt{]} \,. \tag{F.6}$$

This conforms to the mathematical convention and makes the array definition in the code segment above look exactly the same as the original matrix in Eqn. (F.5). Note that in this scheme the first array index corresponds to the *vertical* coordinate and the second index to the *horizontal* coordinate.

However, if an array is used to specify the contents of an *image* $I(u,v)$ or a *filter kernel* $H(i,j)$, we usually assume that the first index (u or i, respectively) is associated with the horizontal x-coordinate and the second index (v bzw. j) with the vertical y-coordinate. For example, if we represent the filter kernel

$$H = \begin{bmatrix} h_{0,0} & h_{1,0} & h_{2,0} \\ h_{0,1} & h_{1,1} & h_{2,1} \\ h_{0,2} & h_{1,2} & h_{2,2} \end{bmatrix} = \begin{bmatrix} -1 & -2 & 0 \\ -2 & 0 & 2 \\ 0 & 2 & 1 \end{bmatrix}$$

as a 2D Java array,

```
double[][] H = {{-1,-2, 0},
                {-2, 0, 2},
                { 0, 2, 1}};
```

then the row and column indexes must be *reversed* in order to access the correct elements. In this case we have the relation

$$h_{i,j} \equiv \texttt{H[}j\texttt{][}i\texttt{]}, \tag{F.7}$$

that is, the ordering of the indexes for array `H` is not the same as for the i/j coordinates of the filter kernel. In this case the *first* array index (j) corresponds to the *vertical* coordinate and the *second* index (i) to the *horizontal* coordinate. The advantage is that (as shown in the aforementioned code segment) the definition of the filter kernel

can be written in the usual matrix form[13] (otherwise we would have to specify the transposed kernel matrix).

If a 2D array is merely used as an image container (whose contents are never defined in matrix form) any convention can be used for the ordering of the indexes. For example, the ImageJ method `getFloatArray()` of class `ImageProcessor`, when called in the form

```
float[][] I = ip.getFloatArray();
```

returns the image as a 2D array (`I`), whose indexes are arranged in the usual x/y order, that is,

$$I(x, y) \equiv \texttt{I[}x\texttt{][}y\texttt{]}. \tag{F.8}$$

In this case, the image pixels are arranged in column-order, that is, *vertically* adjacent elements are stored next to each other in memory.

Size of multi-dimensional arrays

The size of a multi-dimensional array can be obtained by querying the size of its sub-arrays. For example, given the following 3D array with dimensions $P \times Q \times R$,

```
int A[][][] = new int[P][Q][R];
```

the size of `A` along its three dimensions is obtained by the statements

```
int p = A.length;          // = P
int q = A[0].length;       // = Q
int r = A[0][0].length;    // = R
```

This at least works for "rectangular" Java arrays, that is, multi-dimensional arrays with all sub-arrays at the same level having *identical* lengths, which is warranted by the array initialization in the aforementioned case. However, every 1D sub-array of `A` may be replaced by a suitable 1D array of *different* length,[14] for example, by the statement

```
A[0][0] = new int[0];
```

To avoid "index-out-of-bounds" errors, the length of each sub-array should be determined dynamically. The following example shows a "bullet-proof" iteration over all elements of a 3D array `A` whose sub-arrays may have different lengths or may even be empty:

```
int A[][][];
...
for (int i = 0; i < A.length; i++) {
  for (int j = 0; j < A[i].length; j++) {
    for (int k = 0; k < A[i][j].length; k++) {
      // safely access A[i][j][k]
    }
  }
}
```

[13] This scheme is used, for example, in the implementation of the 3×3 filter plugin in Prog. 5.2 (Chapter 5, p. 95).

[14] Even if the array `A` was originally declared `final`, the structure and contents of its sub-arrays may be modified any time.

F.2.5 Arrays of Objects

In Java, as mentioned earlier, we can create arrays dynamically; that is, the size of an array can be specified at runtime. This is convenient because we can adapt the size of the arrays to the given problem. For example, we could write

```
Corner[] corners = new Corner[n];
```

to create an array that can hold `n` objects of type `Corner` (as defined in Chapter 7, Sec. 7.3). Note that the new array `corners` is not filled with corners yet but initialized with `null` references, so the newly created array holds no objects at all. We can insert a `Corner` object into its first (or any other) cell, for example, by

```
corners[0] = new Corner(10, 20, 6789.0f);
```

F.2.6 Searching for Minimum and Maximum Values

Unfortunately, the standard Java API does not provide methods for retrieving the minimum and maximum values of a numeric array. Although these values are easily found by iterating over all elements of the sequence, care must be taken regarding the initialization.

For example, finding the extreme values of a sequence of `int`-values could be accomplished as follows:[15]

```
int[] A = ...
int minval = Integer.MAX_VALUE;
int maxval = Integer.MIN_VALUE;
for (int val : A) {
  minval = Math.min(minval, val);
  maxval = Math.max(maxval, val);
}
```

Note the use of the constants `MIN_VALUE` and `MAX_VALUE`, which are defined for any numeric Java type.

However, in the case of *floating-point* values, these are not the proper values for initialization.[16] Instead, `POSITIVE_INFINITY` and `NEGATIVE_INFINITY` should be used, as shown in the following code segment:

```
double[] B = ...
double minval = Double.POSITIVE_INFINITY;
double maxval = Double.NEGATIVE_INFINITY;
for (double val : B) {
  minval = Math.min(minval, val);
  maxval = Math.max(maxval, val);
}
```

[15] Alternatively, one could initialize `minval` and `maxval` with the first array element `A[0]`.

[16] Because `Double.MIN_VALUE` and `Float.MIN_VALUE` specify to the smallest *positive* values.

F.2.7 Sorting Arrays

Arrays can be sorted efficiently with the standard method

```
Arrays.sort(type[] arr)
```

in class `java.util.Arrays`, where `arr` can be any array of primitive *`type`* (`int`, `float`, etc.) or an array of objects. In the latter case, the array may not have `null` entries. Also, the class of every contained object must implement the `Comparable` interface, that is, provide a public method `compareTo()` that returns an `int` value of -1, 0, or 1, depending upon the intended ordering relation. For example, the class `Corner` defines the `compareTo()` method as follows:

```
public class Corner implements Comparable<Corner> {
  float x, y, q;
  ...
  public int compareTo(Corner other) {
    if (this.q > other.q) return -1;
    else if (this.q < other.q) return 1;
    else return 0;
  }
}
```

References

1. Adobe Systems. "Adobe RGB (1998) Color Space Specification" (2005). http://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf.
2. M. Ahmed and R. Ward. A rotation invariant rule-based thinning algorithm for character recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24**(12), 1672–1678 (2002).
3. L. Alvarez, P.-L. Lions, and J.-M. Morel. Image selective smoothing and edge detection by nonlinear diffusion (II). *SIAM Journal on Numerical Analysis* **29**(3), 845–866 (1992).
4. Apache Software Foundation. "Commons Math: The Apache Commons Mathematics Library". http://commons.apache.org/math/index.html.
5. K. Arbter, W. E. Snyder, H. Burkhardt, and G. Hirzinger. Application of affine-invariant Fourier descriptors to recognition of 3-D objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **12**(7), 640–647 (1990).
6. G. R. Arce, J. Bacca, and J. L. Paredes. Nonlinear filtering for image analysis and enhancement. In A. Bovik, editor, "Handbook of Image and Video Processing", pp. 109–133. Academic Press, New York, second ed. (2005).
7. C. Arcelli and G. Sanniti di Baja. A one-pass two-operation process to detect the skeletal pixels on the 4-distance transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **11**(4), 411–414 (1989).
8. K. Arnold, J. Gosling, and D. Holmes. "The Java Programming Language". Prentice Hall, fifth ed. (2012).
9. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM* **45**(6), 891–923 (1998).
10. J. Astola, P. Haavisto, and Y. Neuvo. Vector median filters. *Proceedings of the IEEE* **78**(4), 678–689 (1990).
11. J. Babaud, A. P. Witkin, M. Baudin, and R. O. Duda. Uniqueness of the Gaussian kernel for scale-space filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(1), 26–33 (1986).
12. W. Bailer. "Writing ImageJ Plugins—A Tutorial" (2003). http://www.imagingbook.com.
13. S. Baker and I. Matthews. Lucas-Kanade 20 years on: A unifying framework: Part 1. Technical Report CMU-RI-TR-02-16, Robotics Institute, Carnegie Mellon University (2003).
14. S. Baker and I. Matthews. Lucas-Kanade 20 years on: A unifying framework. *International Journal of Computer Vision* **56**(3), 221–255 (2004).
15. D. H. Ballard and C. M. Brown. "Computer Vision". Prentice Hall, Englewood Cliffs, NJ (1982).
16. D. Barash. Fundamental relationship between bilateral filtering, adaptive smoothing, and the nonlinear diffusion equation. *IEEE*

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9

Transactions on Pattern Analysis and Machine Intelligence **24**(6), 844–847 (2002).

17. C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software* **22**(4), 469–483 (1996).
18. M. Barni. A fast algorithm for 1-norm vector median filtering. *IEEE Transactions on Image Processing* **6**(10), 1452–1455 (1997).
19. H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf. Parametric correspondence and chamfer matching: two new techniques for image matching. In R. Reddy, editor, "Proceedings of the 5th International Joint Conference on Artificial Intelligence", pp. 659–663, Cambridge, MA (1977). William Kaufmann, Los Altos, CA.
20. H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. SURF: Speeded up robust features. *Computer Vision, Graphics, and Image Processing: Image Understanding* **110**(3), 346–359 (2008).
21. J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In "Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR'97)", pp. 1000–1006, Puerto Rico (June 1997).
22. R. Bencina and M. Kaltenbrunner. The design and evolution of fiducials for the reacTIVision system. In "Proceedings of the 3rd International Conference on Generative Systems in the Electronic Arts", Melbourne (2005).
23. J. Bernsen. Dynamic thresholding of grey-level images. In "Proceedings of the International Conference on Pattern Recognition (ICPR)", pp. 1251–1255, Paris (October 1986). IEEE Computer Society.
24. C. M. Bishop. "Pattern Recognition and Machine Learning". Springer, New York (2006).
25. R. E. Blahut. "Fast Algorithms for Digital Signal Processing". Addison-Wesley, Reading, MA (1985).
26. I. Blayvas, A. Bruckstein, and R. Kimmel. Efficient computation of adaptive threshold surfaces for image binarization. *Pattern Recognition* **39**(1), 89–101 (2006).
27. J. Blinn. Consider the lowly 2×2 matrix. *IEEE Computer Graphics and Applications* **16**(2), 82–88 (1996).
28. J. Blinn. "Jim Blinn's Corner: Notation, Notation, Notation". Morgan Kaufmann (2002).
29. J. Bloch. "Effective Java". Addison-Wesley, second ed. (2008).
30. G. Borgefors. Distance transformations in digital images. *Computer Vision, Graphics and Image Processing* **34**, 344–371 (1986).
31. G. Borgefors. Hierarchical chamfer matching: a parametric edge matching algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **10**(6), 849–865 (1988).
32. A. I. Borisenko and I. E. Tarapov. "Vector and Tensor Analysis with Applications". Dover Publications, New York (1979).
33. J. E. Bresenham. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM* **20**(2), 100–106 (1977).
34. E. O. Brigham. "The Fast Fourier Transform and Its Applications". Prentice Hall, Englewood Cliffs, NJ (1988).
35. I. N. Bronstein and K. A. Semendjajew. "Handbook of Mathematics". Springer-Verlag, Berlin, third ed. (2007).
36. I. N. Bronstein, K. A. Semendjajew, G. Musiol, and H. Mühlig. "Taschenbuch der Mathematik". Verlag Harri Deutsch, fifth ed. (2000).
37. M. Brown and D. Lowe. Invariant features from interest point groups. In "Proceedings of the British Machine Vision Conference", pp. 656–665 (2002).

38. H. Bunke and P. S.-P. Wang, editors. "Handbook of Character Recognition and Document Image Analysis". World Scientific, Singapore (2000).
39. W. Burger and M. J. Burge. "Digital Image Processing—An Algorithmic Introduction using Java". Texts in Computer Science. Springer, New York (2008).
40. W. Burger and M. J. Burge. "ImageJ Short Reference for Java Developers" (2008). http://www.imagingbook.com.
41. P. J. Burt and E. H. Adelson. The Laplacian pyramid as a compact image code. *IEEE Transactions on Communications* **31**(4), 532–540 (1983).
42. J. F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(6), 679–698 (1986).
43. K. R. Castleman. "Digital Image Processing". Prentice Hall, Upper Saddle River, NJ (1995).
44. E. E. Catmull and R. Rom. A class of local interpolating splines. In R. E. Barnhill and R. F. Riesenfeld, editors, "Computer Aided Geometric Design", pp. 317–326. Academic Press, New York (1974).
45. F. Catté, P.-L. Lions, J.-M. Morel, and T. Coll. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM Journal on Numerical Analysis* **29**(1), 182–193 (1992).
46. C. I. Chang, Y. Du, J. Wang, S. M. Guo, and P. D. Thouin. Survey and comparative analysis of entropy and relative entropy thresholding techniques. *IEE Proceedings—Vision, Image and Signal Processing* **153**(6), 837–850 (2006).
47. F. Chang, C. J. Chen, and C. J. Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision, Graphics, and Image Processing: Image Understanding* **93**(2), 206–220 (2004).
48. P. Charbonnier, L. Blanc-Feraud, G. Aubert, and M. Barlaud. Two deterministic half-quadratic regularization algorithms for computed imaging. In "Proceedings IEEE International Conference on Image Processing (ICIP-94)", vol. 2, pp. 168–172, Austin (November 1994).
49. Y. Chen and G. Leedham. Decompose algorithm for thresholding degraded historical document images. *IEE Proceedings—Vision, Image and Signal Processing* **152**(6), 702–714 (2005).
50. H. D. Cheng, X. H. Jiang, Y. Sun, and J. Wang. Color image segmentation: advances and prospects. *Pattern Recognition* **34**(12), 2259–2281 (2001).
51. P. R. Cohen and E. A. Feigenbaum. "The Handbook of Artificial Intelligence". William Kaufmann, Los Altos, CA (1982).
52. B. Coll, J. L. Lisani, and C. Sbert. Color images filtering by anisotropic diffusion. In "Proceedings of the IEEE International Conference on Systems, Signals, and Image Processing (IWSSIP)", pp. 305–308, Chalkida, Greece (2005).
53. D. Comaniciu and P. Meer. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24**(5), 603–619 (2002).
54. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. "Introduction to Algorithms". MIT Press, Cambridge, MA, second ed. (2001).
55. R. L. Cosgriff. Identification of shape. Technical Report 820-11, Antenna Laboratory, Ohio State University, Department of Electrical Engineering, Columbus, Ohio (December 1960).

56. A. Criminisi, I. D. Reid, and A. Zisserman. A plane measuring device. *Image and Vision Computing* **17**(8), 625–634 (1999).
57. T. R. Crimmins. A complete set of Fourier descriptors for two-dimensional shapes. *IEEE Transactions on Systems, Man, and Cybernetics* **12**(6), 848–855 (1982).
58. F. C. Crow. Summed-area tables for texture mapping. *SIGGRAPH Computer Graphics* **18**(3), 207–212 (1984).
59. A. Cumani. Edge detection in multispectral images. *Computer Vision, Graphics and Image Processing* **53**(1), 40–51 (1991).
60. A. Cumani. Efficient contour extraction in color images. In "Proceedings of the Third Asian Conference on Computer Vision", ACCV, pp. 582–589, Hong Kong (January 1998). Springer.
61. L. S. Davis. A survey of edge detection techniques. *Computer Graphics and Image Processing* **4**, 248–270 (1975).
62. R. Deriche. Using Canny's criteria to derive a recursively implemented optimal edge detector. *International Journal of Computer Vision* **1**(2), 167–187 (1987).
63. S. Di Zenzo. A note on the gradient of a multi-image. *Computer Vision, Graphics and Image Processing* **33**(1), 116–125 (1986).
64. R. O. Duda, P. E. Hart, and D. G. Stork. "Pattern Classification". Wiley, New York (2001).
65. F. Durand and J. Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. In "Proceedings of the 29th annual conference on Computer graphics and interactive techniques (SIGGRAPH'02)", pp. 257–266, San Antonio, Texas (July 2002).
66. B. Eckel. "Thinking in Java". Prentice Hall, Englewood Cliffs, NJ, fourth ed. (2006). Earlier versions available online.
67. M. Elad. On the origin of the bilateral filter and ways to improve it. *IEEE Transactions on Image Processing* **11**(10), 1141–1151 (2002).
68. A. Ferreira and S. Ubeda. Computing the medial axis transform in parallel with eight scan operations. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **21**(3), 277–282 (1999).
69. N. I. Fisher. "Statistical Analysis of Circular Data". Cambridge University Press (1995).
70. D. Flanagan. "Java in a Nutshell". O'Reilly, Sebastopol, CA, fifth ed. (2005).
71. L. M. J. Florack, B. M. ter Haar Romeny, J. J. Koenderink, and M. A. Viergever. Scale and the differential structure of images. *Image and Vision Computing* **10**(6), 376–388 (1992).
72. J. Flusser. On the independence of rotation moment invariants. *Pattern Recognition* **33**(9), 1405–1410 (2000).
73. J. Flusser. Moment forms invariant to rotation and blur in arbitrary number of dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **25**(2), 234–246 (2003).
74. J. Flusser, B. Zitova, and T. Suk. "Moments and Moment Invariants in Pattern Recognition". John Wiley & Sons (2009).
75. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. "Computer Graphics: Principles and Practice". Addison-Wesley, Reading, MA, second ed. (1996).
76. A. Ford and A. Roberts. "Colour Space Conversions" (1998). http://www.poynton.com/PDFs/coloureq.pdf.
77. W. Förstner and E. Gülch. A fast operator for detection and precise location of distinct points, corners and centres of circular features. In A. Grün and H. Beyer, editors, "Proceedings, International Society for Photogrammetry and Remote Sensing Intercommission Conference on the Fast Processing of Photogrammetric Data", pp. 281–305, Interlaken (June 1987).

78. D. A. Forsyth and J. Ponce. "Computer Vision—A Modern Approach". Prentice Hall, Englewood Cliffs, NJ (2003).
79. H. Freeman. Computer processing of line drawing images. *ACM Computing Surveys* **6**(1), 57–97 (1974).
80. J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software* **3**(3), 209–226 (1977).
81. D. L. Fritzsche. A systematic method for character recognition. Technical Report 1222-4, Antenna Laboratory, Ohio State University, Department of Electrical Engineering, Columbus, Ohio (November 1961).
82. M. Gervautz and W. Purgathofer. A simple method for color quantization: octree quantization. In A. Glassner, editor, "Graphics Gems I", pp. 287–293. Academic Press, New York (1990).
83. T. Gevers, A. Gijsenij, J. van de Weijer, and J.-M. Geusebroek. "Color in Computer Vision". Wiley (2012).
84. T. Gevers and H. Stokman. Classifying color edges in video into shadow-geometry, highlight, or material transitions. *IEEE Transactions on Multimedia* **5**(2), 237–243 (2003).
85. T. Gevers, J. Van De Weijer, and H. Stokman. Color feature detection. In R. Lukac and K. N. Plataniotis, editors, "Color Image Processing: Methods and Applications", pp. 203–226. CRC Press (2006).
86. C. A. Glasbey. An analysis of histogram-based thresholding algorithms. *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing* **55**(6), 532–537 (1993).
87. A. S. Glassner. "Principles of Digital Image Synthesis". Morgan Kaufmann Publishers, San Francisco (1995).
88. R. C. Gonzalez and R. E. Woods. "Digital Image Processing". Addison-Wesley, Reading, MA (1992).
89. R. C. Gonzalez and R. E. Woods. "Digital Image Processing". Pearson Prentice Hall, Upper Saddle River, NJ, third ed. (2008).
90. M. Grabner, H. Grabner, and H. Bischof. Fast approximated SIFT. In "Proceedings of the 7th Asian Conference of Computer Vision", pp. 918–927 (2006).
91. R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters* **1**, 132–133 (1972).
92. R. L. Graham, D. E. Knuth, and O. Patashnik. "Concrete Mathematics: A Foundation for Computer Science". Addison-Wesley, Reading, MA, second ed. (1994).
93. G. H. Granlund. Fourier preprocessing for hand print character recognition. *IEEE Transactions on Computers* **21**(2), 195–201 (1972).
94. P. Green. Colorimetry and colour differences. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 3, pp. 40–77. Wiley, New York (2002).
95. F. Guichard, L. Moisan, and J.-M. Morel. A review of P.D.E. models in image processing and image analysis. *J. Phys. IV France* **12**(1), 137–154 (2002).
96. W. W. Hager. "Applied Numerical Linear Algebra". Prentice Hall (1988).
97. E. L. Hall. "Computer Image Processing and Recognition". Academic Press, New York (1979).
98. A. Hanbury. Circular statistics applied to colour images. In "Proceedings of the 8th Computer Vision Winter Workshop", pp. 55–60, Valtice, Czech Republic (February 2003).

99. J. C. Hancock. "An Introduction to the Principles of Communication Theory". McGraw-Hill (1961).
100. I. Hannah, D. Patel, and R. Davies. The use of variance and entropic thresholding methods for image segmentation. *Pattern Recognition* **28**(4), 1135–1143 (1995).
101. W. W. Harman. "Principles of the Statistical Theory of Communication". McGraw-Hill (1963).
102. C. G. Harris and M. Stephens. A combined corner and edge detector. In C. J. Taylor, editor, "4th Alvey Vision Conference", pp. 147–151, Manchester (1988).
103. R. Hartley and A. Zisserman. "Multiple View Geometry in Computer Vision". Cambridge University Press, 2 ed. (2013).
104. P. S. Heckbert. Color image quantization for frame buffer display. *Computer Graphics* **16**(3), 297–307 (1982).
105. P. S. Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California, Berkeley, Dept. of Electrical Engineering and Computer Science (1989).
106. R. Hess. An open-source SIFT library. In "Proceedings of the International Conference on Multimedia, MM'10", pp. 1493–1496, Firenze, Italy (October 2010).
107. J. Holm, I. Tastl, L. Hanlon, and P. Hubel. Color processing for digital photography. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 9, pp. 179–220. Wiley, New York (2002).
108. C. M. Holt, A. Stewart, M. Clint, and R. H. Perrott. An improved parallel thinning algorithm. *Communications of the ACM* **30**(2), 156–160 (1987).
109. V. Hong, H. Palus, and D. Paulus. Edge preserving filters on color images. In "Proceedings Int'l Conf. on Computational Science, ICCS", pp. 34–40, Kraków, Poland (2004).
110. B. K. P. Horn. "Robot Vision". MIT-Press, Cambridge, MA (1982).
111. P. V. C. Hough. Method and means for recognizing complex patterns. US Patent 3,069,654 (1962).
112. M. K. Hu. Visual pattern recognition by moment invariants. *IEEE Transactions on Information Theory* **8**, 179–187 (1962).
113. A. Huertas and G. Medioni. Detection of intensity changes with subpixel accuracy using Laplacian-Gaussian masks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(5), 651–664 (1986).
114. R. W. G. Hunt. "The Reproduction of Colour". Wiley, New York, sixth ed. (2004).
115. J. Hutchinson. Culture, communication, and an information age madonna. *IEEE Professional Communications Society Newsletter* **45**(3), 1, 5–7 (2001).
116. J. Illingworth and J. Kittler. Minimum error thresholding. *Pattern Recognition* **19**(1), 41–47 (1986).
117. J. Illingworth and J. Kittler. A survey of the Hough transform. *Computer Vision, Graphics and Image Processing* **44**, 87–116 (1988).
118. International Color Consortium. "Specification ICC.1:2010-12 (Profile Version 4.3.0.0): Image Technology Colour Management—Architecture, Profile Format, and Data Structure" (2010). http://www.color.org.
119. International Electrotechnical Commission, IEC, Geneva. "IEC 61966-2-1: Multimedia Systems and Equipment—Colour Measurement and Management, Part 2-1: Colour Management—Default RGB Colour Space—sRGB" (1999). http://www.iec.ch.
120. International Organization for Standardization, ISO, Geneva. "ISO 13655:1996, Graphic Technology—Spectral Measurement and Colorimetric Computation for Graphic Arts Images" (1996).

121. International Organization for Standardization, ISO, Geneva. "ISO 15076-1:2005, Image Technology Colour Management—Architecture, Profile Format, and Data Structure: Part 1" (2005). Based on ICC.1:2004-10.
122. International Telecommunications Union, ITU, Geneva. "ITU-R Recommendation BT.709-3: Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange" (1998).
123. International Telecommunications Union, ITU, Geneva. "ITU-R Recommendation BT.601-5: Studio Encoding Parameters of Digital Television for Standard 4:3 and Wide-Screen 16:9 Aspect Ratios" (1999).
124. K. Jack. "Video Demystified—A Handbook for the Digital Engineer". LLH Publishing, Eagle Rock, VA, third ed. (2001).
125. B. Jähne. "Practical Handbook on Image Processing for Scientific Applications". CRC Press, Boca Raton, FL (1997).
126. B. Jähne. "Digitale Bildverarbeitung". Springer-Verlag, Berlin, fifth ed. (2002).
127. B. Jähne. "Digital Image Processing". Springer-Verlag, Berlin, sixth ed. (2005).
128. A. K. Jain. "Fundamentals of Digital Image Processing". Prentice Hall, Englewood Cliffs, NJ (1989).
129. R. Jain, R. Kasturi, and B. G. Schunck. "Machine Vision". McGraw-Hill, Boston (1995).
130. Y. Jia and T. Darrell. Heavy-tailed distances for gradient based image descriptors. In "Proceedings of the Twenty-Fifth Annual Conference on Neural Information Processing Systems (NIPS)", Grenada, Spain (December 2011).
131. X. Y. Jiang and H. Bunke. Simple and fast computation of moments. *Pattern Recognition* **24**(8), 801–806 (1991).
132. L. Jin and D. Li. A switching vector median filter based on the CIELAB color space for color image restoration. *Signal Processing* **87**(6), 1345–1354 (2007).
133. J. N. Kapur, P. K. Sahoo, and A. K. C. Wong. A new method for gray-level picture thresholding using the entropy of the histogram. *Computer Vision, Graphics, and Image Processing* **29**, 273–285 (1985).
134. B. Kimia. A large binary image database. Technical Report, LEMS Vision Group, Brown University (2002).
135. J. King. Engineering color at Adobe. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 15, pp. 341–369. Wiley, New York (2002).
136. R. A. Kirsch. Computer determination of the constituent structure of biological images. *Computers in Biomedical Research* **4**, 315–328 (1971).
137. L. Kitchen and A. Rosenfeld. Gray-level corner detection. *Pattern Recognition Letters* **1**, 95–102 (1982).
138. D. E. Knuth. "The Art of Computer Programming, Volume 2: Seminumerical Algorithms". Addison-Wesley, third ed. (1997).
139. J. J. Koenderink. The structure of images. *Biological Cybernetics* **50**(5), 363–370 (1984).
140. A. Koschan and M. A. Abidi. Detection and classification of edges in color images. *IEEE Signal Processing Magazine* **22**(1), 64–73 (2005).
141. A. Koschan and M. A. Abidi. "Digital Color Image Processing". Wiley (2008).
142. P. Kovesi. Arbitrary Gaussian filtering with 25 additions and 5 multiplications per pixel. Technical Report UWA-CSSE-09-002, The

University of Western Australia, School of Computer Science and Software Engineering (2009).

143. F. P. Kuhl and C. R. Giardina. Elliptic Fourier features of a closed contour. *Computer Graphics and Image Processing* **18**(3), 236–258 (1982).
144. M. Kuwahara, K. Hachimura, S. Eiho, and M. Kinoshita. Processing of RI-angiocardiographic image. In K. Preston and M. Onoe, editors, "Digital Processing of Biomedical Images", pp. 187–202. Plenum, New York (1976).
145. D. C. Lay. "Linear Algebra and Its Applications". Pearson, Boston, third ed. (2006).
146. P. E. Lestrel, editor. "Fourier Descriptors and Their Applications in Biology". Cambridge University Press, New York (1997).
147. P.-S. Liao, T.-S. Chen, and P.-C. Chung. A fast algorithm for multilevel thresholding. *Journal of Information Science and Engineering* **17**, 713–727 (2001).
148. C. C. Lin and R. Chellappa. Classification of partial 2-D shapes using Fourier descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **9**(5), 686–690 (1987).
149. B. J. Lindbloom. Accurate color reproduction for computer graphics applications. *SIGGRAPH Computer Graphics* **23**(3), 117–126 (1989).
150. T. Lindeberg. "Scale-Space Theory in Computer Vision". Kluwer Academic Publishers (1994).
151. T. Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision* **30**(2), 77–116 (1998).
152. D. G. Lowe. Object recognition from local scale-invariant features. In "Proceedings of the 7th IEEE International Conference on Computer Vision", vol. 2 of "ICCV'99", pp. 1150–1157, Kerkyra, Corfu, Greece (1999).
153. D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision* **60**, 91–110 (2004).
154. B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In P. J. Hayes, editor, "Proceedings of the 7th International Joint Conference on Artificial Intelligence IJCAI'81", pp. 674–679, Vancouver, BC (1981). William Kaufmann, Los Altos, CA.
155. R. Lukac, B. Smolka, and K. N. Plataniotis. Sharpening vector median filters. *Signal Processing* **87**(9), 2085–2099 (2007).
156. R. Lukac, B. Smolka, K. N. Plataniotis, and A. N. Venetsanopoulos. Vector sigma filters for noise detection and removal in color images. *Journal of Visual Communication and Image Representation* **17**(1), 1–26 (2006).
157. P. C. Mahalanobis. On the generalised distance in statistics. *Proceedings of the National Institute of Sciences of India* **2**(1), 49–55 (1936).
158. S. Mallat. "A Wavelet Tour of Signal Processing". Academic Press, New York (1999).
159. C. Mancas-Thillou and B. Gosselin. Color text extraction with selective metric-based clustering. *Computer Vision, Graphics, and Image Processing: Image Understanding* **107**(1-2), 97–107 (2007).
160. M. J. Maron and R. J. Lopez. "Numerical Analysis". Wadsworth Publishing, third ed. (1990).
161. D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London, Series B* **207**, 187–217 (1980).
162. E. H. W. Meijering, W. J. Niessen, and M. A. Viergever. Quantitative evaluation of convolution-based methods for medical image interpolation. *Medical Image Analysis* **5**(2), 111–126 (2001).

163. J. Miano. "Compressed Image File Formats". ACM Press, Addison-Wesley, Reading, MA (1999).
164. D. P. Mitchell and A. N. Netravali. Reconstruction filters in computer-graphics. In R. J. Beach, editor, "Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'88", pp. 221–228, Atlanta, GA (1988). ACM Press, New York.
165. P. A. Mlsna and J. J. Rodriguez. Gradient and Laplacian-type edge detection. In A. Bovik, editor, "Handbook of Image and Video Processing", pp. 415–431. Academic Press, New York (2000).
166. P. A. Mlsna and J. J. Rodriguez. Gradient and Laplacian-type edge detection. In A. Bovik, editor, "Handbook of Image and Video Processing", pp. 415–431. Academic Press, New York, second ed. (2005).
167. J. Morovic. "Color Gamut Mapping". Wiley (2008).
168. J. D. Murray and W. VanRyper. "Encyclopedia of Graphics File Formats". O'Reilly, Sebastopol, CA, second ed. (1996).
169. M. Nadler and E. P. Smith. "Pattern Recognition Engineering". Wiley, New York (1993).
170. M. Nagao and T. Matsuyama. Edge preserving smoothing. *Computer Graphics and Image Processing* **9**(4), 394–407 (1979).
171. S. K. Naik and C. A. Murthy. Standardization of edge magnitude in color images. *IEEE Transactions on Image Processing* **15**(9), 2588–2595 (2006).
172. W. Niblack. "An Introduction to Digital Image Processing". Prentice-Hall (1986).
173. M. Nitzberg and T. Shiota. Nonlinear image filtering with edge and corner enhancement. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **14**(8), 826–833 (1992).
174. M. Nixon and A. Aguado. "Feature Extraction and Image Processing". Academic Press, second ed. (2008).
175. W. Oh and W. B. Lindquist. Image thresholding by indicator kriging. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **21**(7), 590–602 (1999).
176. A. V. Oppenheim, R. W. Shafer, and J. R. Buck. "Discrete-Time Signal Processing". Prentice Hall, Englewood Cliffs, NJ, second ed. (1999).
177. N. Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics* **9**(1), 62–66 (1979).
178. N. R. Pal and S. K. Pal. A review on image segmentation techniques. *Pattern Recognition* **26**(9), 1277–1294 (1993).
179. S. Paris and F. Durand. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision* **81**(1), 24–52 (2007).
180. T. Pavlidis. "Algorithms for Graphics and Image Processing". Computer Science Press / Springer-Verlag, New York (1982).
181. O. Pele and M. Werman. A linear time histogram metric for improved SIFT matching. In "Proceedings of the 10th European Conference on Computer Vision (ECCV'08)", pp. 495–508, Marseille, France (October 2008).
182. P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **12**(4), 629–639 (1990).
183. E. Persoon and K.-S. Fu. Shape discrimination using Fourier descriptors. *IEEE Transactions on Systems, Man and Cybernetics* **7**(3), 170–179 (1977).

184. E. Persoon and K.-S. Fu. Shape discrimination using Fourier descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **8**(3), 388–397 (1986).
185. T. Q. Pham and L. J. van Vliet. Separable bilateral filtering for fast video preprocessing. In "Proceedings IEEE International Conference on Multimedia and Expo", pp. CD1–4, Los Alamitos, USA (July 2005). IEEE Computer Society.
186. K. N. Plataniotis and A. N. Venetsanopoulos. "Color Image Processing and Applications". Springer (2000).
187. F. Porikli. Constant time O(1) bilateral filtering. In "Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)", pp. 1–8, Anchorage (June 2008).
188. C. A. Poynton. "Digital Video and HDTV Algorithms and Interfaces". Morgan Kaufmann Publishers, San Francisco (2003).
189. S. Prakash and F. V. D. Heyden. Normalisation of Fourier descriptors of planar shapes. *Electronics Letters* **19**(20), 828–830 (1983).
190. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. "Numerical Recipes". Cambridge University Press, third ed. (2007).
191. J. Prewitt. Object enhancement and extraction. In B. Lipkin and A. Rosenfeld, editors, "Picture Processing and Psychopictorics", pp. 415–431. Academic Press (1970).
192. R. R. Rakesh, P. Chaudhuri, and C. A. Murthy. Thresholding in edge detection: a statistical approach. *IEEE Transactions on Image Processing* **13**(7), 927–936 (2004).
193. W. S. Rasband. "ImageJ". U.S. National Institutes of Health, MD (1997–2007). http://rsb.info.nih.gov/ij/.
194. C. E. Reid and T. B. Passin. "Signal Processing in C". Wiley, New York (1992).
195. D. Rich. Instruments and methods for colour measurement. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 2, pp. 19–48. Wiley, New York (2002).
196. C. W. Richard and H. Hemami. Identification of three-dimensional objects using Fourier descriptors of the boundary curve. *IEEE Transactions on Systems, Man, and Cybernetics* **4**(4), 371–378 (1974).
197. I. E. G. Richardson. "H.264 and MPEG-4 Video Compression". Wiley, New York (2003).
198. T. W. Ridler and S. Calvard. Picture thresholding using an iterative selection method. *IEEE Transactions on Systems, Man, and Cybernetics* **8**(8), 630–632 (1978).
199. L. G. Roberts. Machine perception of three-dimensional solids. In J. T. Tippet, editor, "Optical and Electro-Optical Information Processing", pp. 159–197. MIT Press, Cambridge, MA (1965).
200. G. Robinson. Edge detection by compass gradient masks. *Computer Graphics and Image Processing* **6**(5), 492–501 (1977).
201. P. I. Rockett. An improved rotation-invariant thinning algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**(10), 1671–1674 (2005).
202. A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM* **12**, 471–494 (1966).
203. J. C. Russ. "The Image Processing Handbook". CRC Press, Boca Raton, FL, third ed. (1998).
204. P. K. Sahoo, S. Soltani, A. K. C. Wong, and Y. C. Chen. A survey of thresholding techniques. *Computer Vision, Graphics and Image Processing* **41**(2), 233–260 (1988).
205. G. Sapiro. "Geometric Partial Differential Equations and Image Analysis". Cambridge University Press (2001).

206. G. Sapiro and D. L. Ringach. Anisotropic diffusion of multivalued images with applications to color filtering. *IEEE Transactions on Image Processing* **5**(11), 1582–1586 (1996).
207. J. Sauvola and M. Pietikäinen. Adaptive document image binarization. *Pattern Recognition* **33**(2), 1135–1143 (2000).
208. H. Schildt. "Java: A Beginner's Guide". Mcgraw-Hill Osborne Media (2014).
209. C. Schmid and R. Mohr. Local grayvalue invariants for image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **19**(5), 530–535 (1997).
210. C. Schmid, R. Mohr, and C. Bauckhage. Evaluation of interest point detectors. *International Journal of Computer Vision* **37**(2), 151–172 (2000).
211. Y. Schwarzer, editor. "Die Farbenlehre Goethes". Westerweide Verlag, Witten (2004).
212. M. Seul, L. O'Gorman, and M. J. Sammon. "Practical Algorithms for Image Analysis". Cambridge University Press, Cambridge (2000).
213. M. Sezgin and B. Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging* **13**(1), 146–165 (2004).
214. L. G. Shapiro and G. C. Stockman. "Computer Vision". Prentice Hall, Englewood Cliffs, NJ (2001).
215. G. Sharma and H. J. Trussell. Digital color imaging. *IEEE Transactions on Image Processing* **6**(7), 901–932 (1997).
216. F. Y. Shih and S. Cheng. Automatic seeded region growing for color image segmentation. *Image and Vision Computing* **23**(10), 877–886 (2005).
217. N. Silvestrini and E. P. Fischer. "Farbsysteme in Kunst und Wissenschaft". DuMont, Cologne (1998).
218. S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Feature tracking and matching in video using programmable graphics hardware. *Machine Vision and Applications* **22**(1), 207–217 (2011).
219. Y. Sirisathitkul, S. Auwatanamongkol, and B. Uyyanonvara. Color image quantization using distances between adjacent colors along the color axis with highest color variance. *Pattern Recognition Letters* **25**, 1025–1043 (2004).
220. S. M. Smith and J. M. Brady. SUSAN—a new approach to low level image processing. *International Journal of Computer Vision* **23**(1), 45–78 (1997).
221. B. Smolka, M. Szczepanski, K. N. Plataniotis, and A. N. Venetsanopoulos. Fast modified vector median filter. In "Proceedings of the 9th International Conference on Computer Analysis of Images and Patterns", CAIP'01, pp. 570–580, London, UK (2001). Springer-Verlag.
222. M. Sonka, V. Hlavac, and R. Boyle. "Image Processing, Analysis and Machine Vision". PWS Publishing, Pacific Grove, CA, second ed. (1999).
223. M. Spiegel and S. Lipschutz. "Schaum's Outline of Vector Analysis". McGraw-Hill, New York, second ed. (2009).
224. M. Stokes and M. Anderson. "A Standard Default Color Space for the Internet—sRGB". Hewlett-Packard, Microsoft, www.w3.org/Graphics/Color/sRGB.html (1996).
225. S. Süsstrunk. Managing color in digital image libraries. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 17, pp. 385–419. Wiley, New York (2002).
226. B. Tang, G. Sapiro, and V. Caselles. Color image enhancement via chromaticity diffusion. *IEEE Transactions on Image Processing* **10**(5), 701–707 (2001).

227. C.-Y. Tang, Y.-L. Wu, M.-K. Hor, and W.-H. Wang. Modified SIFT descriptor for image matching under interference. In "Proceedings of the International Conference on Machine Learning and Cybernetics (ICMLC)", pp. 3294–3300, Kunming, China (July 2008).
228. S. Theodoridis and K. Koutroumbas. "Pattern Recognition". Academic Press, New York (1999).
229. C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In "Proceedings Int'l Conf. on Computer Vision", ICCV'98, pp. 839–846, Bombay (1998).
230. F. Tomita and S. Tsuji. Extraction of multiple regions by smoothing in selected neighborhoods. *IEEE Transactions on Systems, Man, and Cybernetics* **7**, 394–407 (1977).
231. Ø. D. Trier and T. Taxt. Evaluation of binarization methods for document images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **17**(3), 312–315 (1995).
232. E. Trucco and A. Verri. "Introductory Techniques for 3-D Computer Vision". Prentice Hall, Englewood Cliffs, NJ (1998).
233. D. Tschumperlé. "PDEs Based Regularization of Multivalued Images and Applications". PhD thesis, Université de Nice, Sophia Antipolis, France (2005).
234. D. Tschumperlé. Fast anisotropic smoothing of multi-valued images using curvature-preserving PDEs. *International Journal of Computer Vision* **68**(1), 65–82 (2006).
235. D. Tschumperlé and R. Deriche. Diffusion PDEs on vector-valued images: local approach and geometric viewpoint. *IEEE Signal Processing Magazine* **19**(5), 16–25 (2002).
236. D. Tschumperlé and R. Deriche. Vector-valued image regularization with PDEs: A common framework for different applications. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**(4), 506–517 (2005).
237. K. Turkowski. Filters for common resampling tasks. In A. Glassner, editor, "Graphics Gems I", pp. 147–165. Academic Press, New York (1990).
238. T. Tuytelaars and L. J. Van Gool. Matching widely separated views based on affine invariant regions. *International Journal of Computer Vision* **59**(1), 61–85 (2004).
239. J. van de Weijer. "Color Features and Local Structure in Images". PhD thesis, University of Amsterdam (2005).
240. M. I. Vardavoulia, I. Andreadis, and P. Tsalides. A new vector median filter for colour image processing. *Pattern Recognition Letters* **22**(6-7), 675–689 (2001).
241. A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. http://www.vlfeat.org/ (2008).
242. F. R. D. Velasco. Thresholding using the ISODATA clustering algorithm. *IEEE Transactions on Systems, Man, and Cybernetics* **10**(11), 771–774 (1980).
243. D. Vernon. "Machine Vision". Prentice Hall (1999).
244. P. Viola and M. Jones. Robust real-time face detection. *International Journal of Computer Vision* **57**(2), 137–154 (2004).
245. T. P. Wallace and P. A. Wintz. An efficient three-dimensional aircraft recognition algorithm using normalized Fourier descriptors. *Computer Vision, Graphics and Image Processing* **13**(2), 99–126 (1980).
246. D. Wallner. Color management and transformation through ICC profiles. In P. Green and L. MacDonald, editors, "Colour Engineering", ch. 11, pp. 247–261. Wiley, New York (2002).

247. A. Watt. “3D Computer Graphics”. Addison-Wesley, Reading, MA, third ed. (1999).
248. A. Watt and F. Policarpo. “The Computer Image”. Addison-Wesley, Reading, MA (1999).
249. J. Weickert. “Anisotropic Diffusion in Image Processing”. PhD thesis, Universität Kaiserslautern, Fachbereich Mathematik (1996).
250. J. Weickert. A review of nonlinear diffusion filtering. In B. M. ter Haar Romeny, L. Florack, J. J. Koenderink, and M. A. Viergever, editors, “Proceedings First International Conference on Scale-Space Theory in Computer Vision, Scale-Space’97”, Lecture Notes in Computer Science, pp. 3–28, Utrecht (July 1997). Springer.
251. J. Weickert. Coherence-enhancing diffusion filtering. *International Journal of Computer Vision* **31**(2/3), 111–127 (1999).
252. J. Weickert. Coherence-enhancing diffusion of colour images. *Image and Vision Computing* **17**(3/4), 201–212 (1999).
253. B. Weiss. Fast median and bilateral filtering. *ACM Transactions on Graphics* **25**(3), 519–526 (2006).
254. M. Welk, J. Weickert, F. Becker, C. Schnörr, C. Feddern, and B. Burgeth. Median and related local filters for tensor-valued images. *Signal Processing* **87**(2), 291–308 (2007).
255. P. Wendykier. “High Performance Java Software for Image Processing”. PhD thesis, Emory University (2009).
256. G. Wolberg. “Digital Image Warping”. IEEE Computer Society Press, Los Alamitos, CA (1990).
257. M.-F. Wu and H.-T. Sheu. Contour-based correspondence using Fourier descriptors. *IEE Proceedings—Vision, Image and Signal Processing* **144**(3), 150–160 (1997).
258. G. Wyszecki and W. S. Stiles. “Color Science: Concepts and Methods, Quantitative Data and Formulae”. Wiley–Interscience, New York, second ed. (2000).
259. Q. Yang, K.-H. Tan, and N. Ahuja. Real-time O(1) bilateral filtering. In “Proceedings IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)”, pp. 557–564, Miami (2009).
260. S. D. Yanowitz and A. M. Bruckstein. A new method for image segmentation. *Computer Vision, Graphics, and Image Processing* **46**(1), 82–95 (1989).
261. G. W. Zack, W. E. Rogers, and S. A. Latt. Automatic measurement of sister chromatid exchange frequency. *Journal of Histochemistry and Cytochemistry* **25**(7), 741–753 (1977).
262. C. T. Zahn and R. Z. Roskies. Fourier descriptors for plane closed curves. *IEEE Transactions on Computers* **21**(3), 269–281 (1972).
263. P. Zamperoni. A note on the computation of the enclosed area for contour-coded binary objects. *Signal Processing* **3**(3), 267–271 (1981).
264. E. Zeidler, editor. “Teubner-Taschenbuch der Mathematik”. B. G. Teubner Verlag, Leipzig, second ed. (2002).
265. T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Communications of the ACM* **27**(3), 236–239 (1984).
266. S.-Y. Zhu, K. N. Plataniotis, and A. N. Venetsanopoulos. Comprehensive analysis of edge detection in color image processing. *Optical Engineering* **38**(4), 612–625 (1999).
267. S. Zokai and G. Wolberg. Image registration using log-polar mappings for recovery of large-scale similarity and projective transformations. *IEEE Transactions on Image Processing* **14**(10), 1422–1434 (2005).

Index

W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9

B

C

D

G

H

I

M

N

Q

R

S

T

U

V

W

X

About the Authors

Wilhelm Burger received a Master's degree in Computer Science from the University of Utah (Salt Lake City) and a doctorate in Systems Science from Johannes Kepler University in Linz, Austria. As a post-graduate researcher at the Honeywell Systems & Research Center in Minneapolis and the University of California at Riverside, he worked mainly in the areas of visual motion analysis and autonomous navigation. Since 1996, he has been Head of the Digital Media Department at the University of Applied Sciences in Hagenberg, Austria. Privately, Wilhelm appreciates large-engine vehicles, chamber music, and (occasionally) a glass of dry "Veltliner".

Mark J. Burge is a senior scientist at Noblis, Inc. in Washington, D.C. He spent seven years as a research scientist with the Swiss Federal Institute of Science (ETH) in Zürich and the Johannes Kepler University in Linz, Austria. He earned tenure as a computer science professor in the University System of Georgia (USG), and later served as a program director at the National Science Foundation (NSF), at MITRE and the Intelligence Advanced Research Programs Activity (IARPA). He also lectures at the United States Naval Academy (USNA). Personally, Mark is an expert on classic Italian espresso machines.