Appendix A

Linear algebra and numerical techniques

A.1 Matrix decompositions . . . 736
A.1.1 Singular value decomposition . . . 736
A.1.2 Eigenvalue decomposition . . . 737
A.1.3 QR factorization . . . 740
A.1.4 Cholesky factorization . . . 741
A.2 Linear least squares . . . 742
A.2.1 Total least squares . . . 744
A.3 Non-linear least squares . . . 746
A.4 Direct sparse matrix techniques . . . 747
A.4.1 Variable reordering . . . 748
A.5 Iterative techniques . . . 748
A.5.1 Conjugate gradient . . . 749
A.5.2 Preconditioning . . . 751
A.5.3 Multigrid . . . 753

In this appendix, we introduce some elements of linear algebra and numerical techniques that are used elsewhere in the book. We start with some basic decompositions in matrix algebra, including the singular value decomposition (SVD), eigenvalue decompositions, and other matrix decompositions (factorizations). Next, we look at the problem of linear least squares, which can be solved using either the QR decomposition or normal equations. This is followed by non-linear least squares, which arise when the measurement equations are not linear in the unknowns or when robust error functions are used. Such problems require iteration to find a solution. Next, we look at direct solution (factorization) techniques for sparse problems, where the ordering of the variables can have a large influence on the computation and memory requirements. Finally, we discuss iterative techniques for solving large linear (or linearized) least squares problems. Good general references for much of this material include the work by Björck (1996), Golub and Van Loan (1996), Trefethen and Bau (1997), Meyer (2000), Nocedal and Wright (2006), and Björck and Dahlquist (2010).

A note on vector and matrix indexing. To be consistent with the rest of the book and with the general usage in the computer science and computer vision communities, I adopt a 0-based indexing scheme for vector and matrix element indexing. Please note that most mathematical textbooks and papers use 1-based indexing, so you need to be aware of the differences when you read this book.

Software implementations. Highly optimized and tested libraries corresponding to the algorithms described in this appendix are readily available and are listed in Appendix C.2.

A.1 Matrix decompositions

In order to better understand the structure of matrices and more stably perform operations such as inversion and system solving, a number of decompositions (or factorizations) can be used. In this section, we review singular value decomposition (SVD), eigenvalue decomposition, QR factorization, and Cholesky factorization.

A.1.1 Singular value decomposition

One of the most useful decompositions in matrix algebra is the *singular value decomposition* (SVD), which states that any real-valued $M \times N$ matrix $\boldsymbol{A}$ can be written as

$$\boldsymbol{A}_{M\times N} = \boldsymbol{U}_{M\times P}\boldsymbol{\Sigma}_{P\times P}\boldsymbol{V}^T_{P\times N} \tag{A.1}$$

$$= \left[\begin{array}{c|c|c} & & \\ \boldsymbol{u}_0 & \cdots & \boldsymbol{u}_{p-1} \\ & & \end{array}\right] \left[\begin{array}{ccc} \sigma_0 & & \\ & \ddots & \\ & & \sigma_{p-1} \end{array}\right] \left[\begin{array}{c} \boldsymbol{v}_0^T \\ \hline \cdots \\ \hline \boldsymbol{v}_{p-1}^T \end{array}\right],$$

where $P = \min(M, N)$. The matrices $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthonormal, i.e., $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$ and $\boldsymbol{V}^T\boldsymbol{V} = \boldsymbol{I}$, and so are their column vectors,

$$\boldsymbol{u}_i \cdot \boldsymbol{u}_j = \boldsymbol{v}_i \cdot \boldsymbol{v}_j = \delta_{ij}. \tag{A.2}$$

The singular values are all non-negative and can be ordered in decreasing order

$$\sigma_0 \geq \sigma_1 \geq \cdots \geq \sigma_{p-1} \geq 0. \tag{A.3}$$

A geometric intuition for the SVD of a matrix $\boldsymbol{A}$ can be obtained by re-writing $\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ in (A.2) as

$$\boldsymbol{A}\boldsymbol{V} = \boldsymbol{U}\boldsymbol{\Sigma} \quad \text{or} \quad \boldsymbol{A}\boldsymbol{v}_j = \sigma_j \boldsymbol{u}_j. \tag{A.4}$$

This formula says that the matrix $\boldsymbol{A}$ takes any basis vector $\boldsymbol{v}_j$ and maps it to a direction $\boldsymbol{u}_j$ with length σ_j, as shown in Figure A.1

If only the first r singular values are positive, the matrix $\boldsymbol{A}$ is of *rank* r and the index p in the SVD decomposition (A.2) can be replaced by r. (In other words, we can drop the last $p - r$ columns of $\boldsymbol{U}$ and $\boldsymbol{V}$.)

An important property of the singular value decomposition of a matrix (also true for the eigenvalue decomposition of a real symmetric non-negative definite matrix) is that if we truncate the expansion

$$\boldsymbol{A} = \sum_{j=0}^{t} \sigma_j \boldsymbol{u}_j \boldsymbol{v}_j^T, \tag{A.5}$$

we obtain the best possible least squares approximation to the original matrix $\boldsymbol{A}$. This is used both in eigenface-based face recognition systems (Section 14.2.1) and in the separable approximation of convolution kernels (3.21).

A.1.2 Eigenvalue decomposition

If the matrix $\boldsymbol{C}$ is symmetric ($m = n$),[1] it can be written as an eigenvalue decomposition,

$$\begin{aligned} \boldsymbol{C} &= \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T = \left[\begin{array}{c|c|c} & & \\ \boldsymbol{u}_0 & \cdots & \boldsymbol{u}_{n-1} \\ & & \end{array}\right] \left[\begin{array}{ccc} \lambda_0 & & \\ & \ddots & \\ & & \lambda_{n-1} \end{array}\right] \left[\begin{array}{c} \boldsymbol{u}_0^T \\ \hline \cdots \\ \hline \boldsymbol{u}_{n-1}^T \end{array}\right] \\ &= \sum_{i=0}^{n-1} \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^T. \end{aligned} \tag{A.6}$$

[1] In this appendix, we denote symmetric matrices using $\boldsymbol{C}$ and general rectangular matrices using $\boldsymbol{A}$.

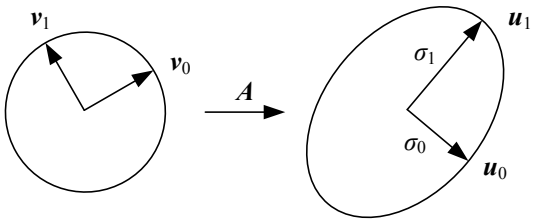

Figure A.1 The action of a matrix $\boldsymbol{A}$ can be visualized by thinking of the domain as being spanned by a set of orthonormal vectors $\boldsymbol{v}_j$, each of which is transformed to a new orthogonal vector $\boldsymbol{u}_j$ with a length σ_j. When $\boldsymbol{A}$ is interpreted as a covariance matrix and its eigenvalue decomposition is performed, each of the $\boldsymbol{u}_j$ axes denote a principal direction (component) and each σ_j denotes one standard deviation along that direction.

(The eigenvector matrix $\boldsymbol{U}$ is sometimes written as $\boldsymbol{\Phi}$ and the eigenvectors $\boldsymbol{u}$ as $\boldsymbol{\phi}$.) In this case, the eigenvalues

$$\lambda_0 \geq \lambda_1 \geq \cdots \geq \lambda_{n-1} \tag{A.7}$$

can be both positive and negative.[2]

A special case of the symmetric matrix $\boldsymbol{C}$ occurs when it is constructed as the sum of a number of outer products

$$\boldsymbol{C} = \sum_i \boldsymbol{a}_i \boldsymbol{a}_i^T = \boldsymbol{A}\boldsymbol{A}^T, \tag{A.8}$$

which often occurs when solving least squares problems (Appendix A.2), where the matrix $\boldsymbol{A}$ consists of all the $\boldsymbol{a}_i$ column vectors stacked side-by-side. In this case, we are guaranteed that all of the eigenvalues λ_i are non-negative. The associated matrix $\boldsymbol{C}$ is *positive semi-definite*

$$\boldsymbol{x}^T \boldsymbol{C} \boldsymbol{x} \geq 0, \quad \forall \boldsymbol{x}. \tag{A.9}$$

If the matrix $\boldsymbol{C}$ is of full rank, the eigenvalues are all positive and the matrix is called *symmetric positive definite* (SPD).

Symmetric positive semi-definite matrices also arise in the statistical analysis of data, since they represent the *covariance* of a set of $\{\boldsymbol{x}_i\}$ points around their mean $\bar{\boldsymbol{x}}$,

$$\boldsymbol{C} = \frac{1}{n} \sum_i (\boldsymbol{x}_i - \bar{\boldsymbol{x}})(\boldsymbol{x}_i - \bar{\boldsymbol{x}})^T. \tag{A.10}$$

In this case, performing the eigenvalue decomposition is known as *principal component analysis* (PCA), since it models the principal directions (and magnitudes) of variation of the point

[2] Eigenvalue decompositions can be computed for non-symmetric matrices but the eigenvalues and eigenvectors can have complex entries in that case.

distribution around their mean, as shown in Section 5.1.1 (5.13–5.15), Section 14.2.1 (14.9), and Appendix B.1.1 (B.10). Figure A.1 shows how the principal components of the covariance matrix $\boldsymbol{C}$ denote the principal axes $\boldsymbol{u}_j$ of the uncertainty ellipsoid corresponding to this point distribution and how the $\sigma_j = \sqrt{\lambda_j}$ denote the standard deviations along each axis.

The eigenvalues and eigenvectors of $\boldsymbol{C}$ and the singular values and singular vectors of $\boldsymbol{A}$ are closely related. Given

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T, \tag{A.11}$$

we get

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T\boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^T = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T. \tag{A.12}$$

From this, we see that $\lambda_i = \sigma_i^2$ and that the left singular vectors of $\boldsymbol{A}$ are the eigenvectors of $\boldsymbol{C}$.

This relationship gives us an efficient method for computing the eigenvalue decomposition of large matrices that are rank deficient, such as the scatter matrices observed in computing eigenfaces (Section 14.2.1). Observe that the covariance matrix $\boldsymbol{C}$ in (14.9) is exactly the same as $\boldsymbol{C}$ in (A.8). Note also that the individual difference-from-mean images $\boldsymbol{a}_i = \boldsymbol{x}_i - \bar{\boldsymbol{x}}$ are long vectors of length P (the number of pixels in the image), while the total number of exemplars N (the number of faces in the training database) is much smaller. Instead of forming $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{A}^T$, which is $P \times P$, we form the matrix

$$\hat{\boldsymbol{C}} = \boldsymbol{A}^T\boldsymbol{A}, \tag{A.13}$$

which is $N \times N$. (This involves taking the dot product between every pair of difference images $\boldsymbol{a}_i$ and $\boldsymbol{a}_j$.) The eigenvalues of $\hat{\boldsymbol{C}}$ are the squared singular values of $\boldsymbol{A}$, namely $\boldsymbol{\Sigma}^2$, and are hence also the eigenvalues of $\boldsymbol{C}$. The eigenvectors of $\hat{\boldsymbol{C}}$ are the right singular vectors $\boldsymbol{V}$ of $\boldsymbol{A}$, from which the desired eigenfaces $\boldsymbol{U}$, which are the left singular vectors of $\boldsymbol{A}$, can be computed as

$$\boldsymbol{U} = \boldsymbol{A}\boldsymbol{V}\boldsymbol{\Sigma}^{-1}. \tag{A.14}$$

This final step is essentially computing the eigenfaces as linear combinations of the difference images (Turk and Pentland 1991a). If you have access to a high-quality linear algebra package such as LAPACK, routines for efficiently computing a small number of the left singular vectors and singular values of rectangular matrices such as $\boldsymbol{A}$ are usually provided (Appendix C.2). However, if storing all of the images in memory is prohibitive, the construction of $\hat{\boldsymbol{C}}$ in (A.13) can be used instead.

How can eigenvalue and singular value decompositions actually be computed? Notice that an eigenvector is defined by the equation

$$\lambda_i\boldsymbol{u}_i = \boldsymbol{C}\boldsymbol{u}_i \quad \text{or} \quad (\lambda_i\boldsymbol{I} - \boldsymbol{C})\boldsymbol{u}_i = 0. \tag{A.15}$$

(This can be derived from (A.6) by post-multiplying both sides by $\boldsymbol{u}_i$.) Since the latter equation is *homogeneous*, i.e., it has a zero right-hand-side, it can only have a non-zero (non-trivial) solution for $\boldsymbol{u}_i$ if the system is rank deficient, i.e.,

$$|(\lambda \boldsymbol{I} - \boldsymbol{C})| = 0. \tag{A.16}$$

Evaluating this determinant yields a *characteristic* polynomial equation in λ, which can be solved for small problems, e.g., 2×2 or 3×3 matrices, in closed form.

For larger matrices, iterative algorithms that first reduce the matrix $\boldsymbol{C}$ to a real symmetric tridiagonal form using orthogonal transforms and then perform QR iterations are normally used (Golub and Van Loan 1996; Trefethen and Bau 1997; Björck and Dahlquist 2010). Since these techniques are rather involved, it is best to use a linear algebra package such as LAPACK (Anderson, Bai, Bischof *et al.* 1999)—see Appendix C.2.

Factorization with missing data requires different kinds of iterative algorithms, which often involve either hallucinating the missing terms or minimizing some weighted reconstruction metric, which is intrinsically much more challenging than regular factorization. This area has been widely studied in computer vision (Shum, Ikeuchi, and Reddy 1995; De la Torre and Black 2003; Huynh, Hartley, and Heyden 2003; Buchanan and Fitzgibbon 2005; Gross, Matthews, and Baker 2006; Torresani, Hertzmann, and Bregler 2008) and is sometimes called *generalized PCA*. However, this term is also sometimes used to denote algebraic subspace clustering techniques, which is the subject of a forthcoming monograph by Vidal, Ma, and Sastry (2010).

A.1.3 QR factorization

A widely used technique for stably solving poorly conditioned least squares problems (Björck 1996) and as the basis of more complex algorithms, such as computing the SVD and eigenvalue decompositions, is the QR factorization,

$$\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}, \tag{A.17}$$

where $\boldsymbol{Q}$ is an *orthonormal* (or *unitary*) matrix $\boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{I}$ and $\boldsymbol{R}$ is upper triangular.[3] In computer vision, QR can be used to convert a camera matrix into a rotation matrix and an upper-triangular calibration matrix (6.35) and also in various self-calibration algorithms (Section 7.2.2). The most common algorithms for computing QR decompositions, modified Gram–Schmidt, Householder transformations, and Givens rotations, are described by Golub and Van Loan (1996), Trefethen and Bau (1997), and Björck and Dahlquist (2010) and are

[3] The term "R" comes from the German name for the lower–upper (LU) decomposition, which is LR for "links" and "rechts" (left and right of the diagonal).

procedure *Cholesky*$(\boldsymbol{C}, \boldsymbol{R})$:

$\boldsymbol{R} = \boldsymbol{C}$

for $i = 0 \ldots n-1$

for $j = i+1 \ldots n-1$

$\boldsymbol{R}_{j,j:n-1} = \boldsymbol{R}_{j,j:n-1} - r_{ij} r_{ii}^{-1} \boldsymbol{R}_{i,j:n-1}$

$\boldsymbol{R}_{i,i:n-1} = r_{ii}^{-1/2} \boldsymbol{R}_{i,i:n-1}$

Algorithm A.1 Cholesky decomposition of the matrix $\boldsymbol{C}$ into its upper triangular form $\boldsymbol{R}$.

also found in LAPACK. Unlike the SVD and eigenvalue decompositions, QR factorization does not require iteration and can be computed exactly in $O(MN^2 + N^3)$ operations, where M is the number of rows and N is the number of columns (for a tall matrix).

A.1.4 Cholesky factorization

Cholesky factorization can be applied to any symmetric positive definite matrix $\boldsymbol{C}$ to convert it into a product of symmetric lower and upper triangular matrices,

$$\boldsymbol{C} = \boldsymbol{L}\boldsymbol{L}^T = \boldsymbol{R}^T\boldsymbol{R}, \tag{A.18}$$

where $\boldsymbol{L}$ is a lower-triangular matrix and $\boldsymbol{R}$ is an upper-triangular matrix. Unlike Gaussian elimination, which may require pivoting (row and column reordering) or may become unstable (sensitive to roundoff errors or reordering), Cholesky factorization remains stable for positive definite matrices, such as those that arise from normal equations in least squares problems (Appendix A.2). Because of the form of (A.18), the matrices $\boldsymbol{L}$ and $\boldsymbol{R}$ are sometimes called *matrix square roots*.[4]

The algorithm to compute an upper triangular Cholesky decomposition of $\boldsymbol{C}$ is a straightforward symmetric generalization of Gaussian elimination and is based on the decomposition (Björck 1996; Golub and Van Loan 1996)

$$\boldsymbol{C} = \begin{bmatrix} \gamma & \boldsymbol{c}^T \\ \boldsymbol{c} & \boldsymbol{C}_{11} \end{bmatrix} \tag{A.19}$$

$$= \begin{bmatrix} \gamma^{1/2} & \boldsymbol{0}^T \\ \boldsymbol{c}\gamma^{-1/2} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} 1 & \boldsymbol{0}^T \\ \boldsymbol{0} & \boldsymbol{C}_{11} - \boldsymbol{c}\gamma^{-1}\boldsymbol{c}^T \end{bmatrix} \begin{bmatrix} \gamma^{1/2} & \gamma^{-1/2}\boldsymbol{c}^T \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \tag{A.20}$$

[4] In fact, there exists a whole family of matrix square roots. Any matrix of the form $\boldsymbol{L}\boldsymbol{Q}$ or $\boldsymbol{Q}\boldsymbol{R}$, where $\boldsymbol{Q}$ is a unitary matrix, is a square root of $\boldsymbol{C}$.

$$= \boldsymbol{R}_0^T \boldsymbol{C}_1 \boldsymbol{R}_0, \tag{A.21}$$

which, through recursion, can be turned into

$$\boldsymbol{C} = \boldsymbol{R}_0^T \ldots \boldsymbol{R}_{n-1}^T \boldsymbol{R}_{n-1} \ldots \boldsymbol{R}_0 = \boldsymbol{R}^T \boldsymbol{R}. \tag{A.22}$$

Algorithm A.1 provides a more procedural definition, which can store the upper-triangular matrix $\boldsymbol{R}$ in the same space as $\boldsymbol{C}$, if desired. The total operation count for Cholesky factorization is $O(N^3)$ for a dense matrix but can be significantly lower for sparse matrices with low fill-in (Appendix A.4).

Note that Cholesky decomposition can also be applied to block-structured matrices, where the term γ in (A.19) is now a square block sub-matrix and $\boldsymbol{c}$ is a rectangular matrix (Golub and Van Loan 1996). The computation of square roots can be avoided by leaving the γ on the diagonal of the middle factor in (A.20), which results in the $\boldsymbol{C} = \boldsymbol{L}\boldsymbol{D}\boldsymbol{L}^T$ factorization, where $\boldsymbol{D}$ is a diagonal matrix. However, since square roots are relatively fast on modern computers, this is not worth the bother and Cholesky factorization is usually preferred.

A.2 Linear least squares

Least squares fitting problems are pervasive in computer vision. For example, the alignment of images based on matching feature points involves the minimization of a squared distance objective function (6.2),

$$E_{\mathrm{LS}} = \sum_i \|\boldsymbol{r}_i\|^2 = \sum_i \|\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{p}) - \boldsymbol{x}'_i\|^2, \tag{A.23}$$

where

$$\boldsymbol{r}_i = \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{p}) - \boldsymbol{x}'_i = \hat{\boldsymbol{x}}'_i - \tilde{\boldsymbol{x}}'_i \tag{A.24}$$

is the *residual* between the measured location $\hat{\boldsymbol{x}}'_i$ and its corresponding current *predicted* location $\tilde{\boldsymbol{x}}'_i = \boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{p})$. More complex versions of least squares problems, such as large-scale structure from motion (Section 7.4), may involve the minimization of functions of thousands of variables. Even problems such as image filtering (Section 3.4.3) and regularization (Section 3.7.1) may involve the minimization of sums of squared errors.

Figure A.2a shows an example of a simple least squares line fitting problem, where the quantities being estimated are the line equation parameters (m, b). When the sampled vertical values y_i are assumed to be noisy versions of points on the line $y = mx + b$, the optimal estimates for (m, b) can be found by minimizing the squared vertical residuals

$$E_{\mathrm{VLS}} = \sum_i |y_i - (mx_i + b)|^2. \tag{A.25}$$

Note that the function being fitted need not itself be linear to use linear least squares. All that is required is that the function be linear in the unknown parameters. For example, polynomial fitting can be written as

$$E_{\mathrm{PLS}} = \sum_i |y_i - (\sum_{j=0}^{p} a_j x_i^j)|^2, \tag{A.26}$$

while sinusoid fitting with unknown amplitude A and phase ϕ (but known frequency f) can be written as

$$E_{\mathrm{SLS}} = \sum_i |y_i - A\sin(2\pi f x_i + \phi)|^2 = \sum_i |y_i - (B\sin 2\pi f x_i + C\cos 2\pi f x_i)|^2, \tag{A.27}$$

which is linear in (B, C).

In general, it is more common to denote the unknown parameters using $\boldsymbol{x}$ and to write the general form of linear least squares as[5]

$$E_{\mathrm{LLS}} = \sum_i |\boldsymbol{a}_i \boldsymbol{x} - b_i|^2 = \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|^2. \tag{A.28}$$

Expanding the above equation gives us

$$E_{\mathrm{LLS}} = \boldsymbol{x}^T(\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{x} - 2\boldsymbol{x}^T(\boldsymbol{A}^T\boldsymbol{b}) + \|\boldsymbol{b}\|^2, \tag{A.29}$$

whose minimum value for $\boldsymbol{x}$ can be found by solving the associated *normal equations* (Björck 1996; Golub and Van Loan 1996)

$$(\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{x} = \boldsymbol{A}^T\boldsymbol{b}. \tag{A.30}$$

The preferred way to solve the normal equations is to use Cholesky factorization. Let

$$\boldsymbol{C} = \boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{R}^T\boldsymbol{R}, \tag{A.31}$$

where $\boldsymbol{R}$ is the upper-triangular Cholesky factor of the Hessian $\boldsymbol{C}$, and

$$\boldsymbol{d} = \boldsymbol{A}^T\boldsymbol{b}. \tag{A.32}$$

After factorization, the solution for $\boldsymbol{x}$ can be obtained as

$$\boldsymbol{R}^T\boldsymbol{z} = \boldsymbol{d}, \quad \boldsymbol{R}\boldsymbol{x} = \boldsymbol{z}, \tag{A.33}$$

which involves the solution of two triangular systems, i.e., forward and backward substitution (Björck 1996).

[5] Be extra careful in interpreting the variable names here. In the 2D line-fitting example, x is used to denote the horizontal axis, but in the general least squares problem, $\boldsymbol{x} = (m, b)$ denotes the unknown parameter vector.

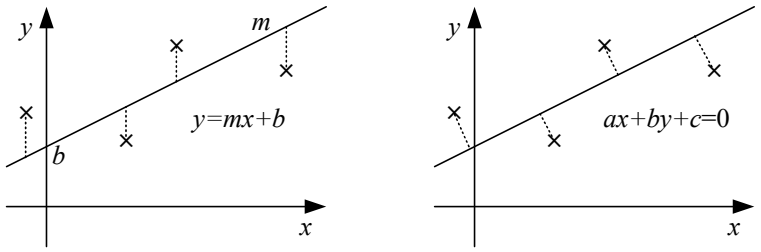

Figure A.2 Least squares regression. (a) The line $y = mx + b$ is fit to the four noisy data points, $\{(x_i, y_i)\}$, denoted by $\times$ by minimizing the squared vertical residuals between the data points and the line, $\sum_i \|y_i - (mx_i + b)\|^2$. (b) When the measurements $\{(x_i, y_i)\}$ are assumed to have noise in all directions, the sum of orthogonal squared distances to the line $\sum_i \|ax_i + by_i + c\|^2$ is minimized using total least squares.

In cases where the least squares problem is numerically poorly conditioned (which should generally be avoided by adding sufficient regularization or prior knowledge about the parameters, (Appendix A.3)), it is possible to use QR factorization or SVD directly on the matrix $\boldsymbol{A}$ (Björck 1996; Golub and Van Loan 1996; Trefethen and Bau 1997; Nocedal and Wright 2006; Björck and Dahlquist 2010), e.g.,

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{Q}\boldsymbol{R}\boldsymbol{x} = \boldsymbol{b} \quad \longrightarrow \quad \boldsymbol{R}\boldsymbol{x} = \boldsymbol{Q}^T\boldsymbol{b}. \tag{A.34}$$

Note that the upper triangular matrices $\boldsymbol{R}$ produced by the Cholesky factorization of $\boldsymbol{C} = \boldsymbol{A}^T\boldsymbol{A}$ and the QR factorization of $\boldsymbol{A}$ are the same, but that solving (A.34) is generally more stable (less sensitive to roundoff error) but slower (by a constant factor).

A.2.1 Total least squares

In some problems, e.g., when performing geometric line fitting in 2D images or 3D plane fitting to point cloud data, instead of having measurement error along one particular axis, the measured points have uncertainty in all directions, which is known as the *errors-in-variables* model (Van Huffel and Lemmerling 2002; Matei and Meer 2006). In this case, it makes more sense to minimize a set of homogeneous squared errors of the form

$$E_{\text{TLS}} = \sum_i (\boldsymbol{a}_i\boldsymbol{x})^2 = \|\boldsymbol{A}\boldsymbol{x}\|^2, \tag{A.35}$$

which is known as *total least squares* (TLS) (Van Huffel and Vandewalle 1991; Björck 1996; Golub and Van Loan 1996; Van Huffel and Lemmerling 2002).

The above error metric has a trivial minimum solution at $\boldsymbol{x} = 0$ and is, in fact, homogeneous in $\boldsymbol{x}$. For this reason, we augment this minimization problem with the requirement that $\|\boldsymbol{x}\|^2 = 1$. which results in the eigenvalue problem

$$\boldsymbol{x} = \arg\min_{\boldsymbol{x}} \boldsymbol{x}^T(\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{x} \quad \text{such that} \quad \|\boldsymbol{x}\|^2 = 1. \tag{A.36}$$

The value of $\boldsymbol{x}$ that minimizes this constrained problem is the eigenvector associated with the smallest eigenvalue of $\boldsymbol{A}^T\boldsymbol{A}$. This is the same as the last right singular vector of $\boldsymbol{A}$, since

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}, \tag{A.37}$$

$$\boldsymbol{A}^T\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}, \tag{A.38}$$

$$\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{v}_k = \sigma_k^2, \tag{A.39}$$

which is minimized by selecting the smallest σ_k value.

Figure A.2b shows a line fitting problem where, in this case, the measurement errors are assumed to be isotropic in (x, y). The solution for the best line equation $ax + by + c = 0$ is found by minimizing

$$E_{\text{TLS}-\text{2D}} = \sum_i (ax_i + by_i + c)^2, \tag{A.40}$$

i.e., finding the eigenvector associated with the smallest eigenvalue of[6]

$$\boldsymbol{C} = \boldsymbol{A}^T\boldsymbol{A} = \sum_i \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \begin{bmatrix} x_i & y_i & 1 \end{bmatrix}. \tag{A.41}$$

Notice, however, that minimizing $\sum_i (\boldsymbol{a}_i\boldsymbol{x})^2$ in (A.35) is only statistically optimal (Appendix B.1.1) if all of the measured terms in the $\boldsymbol{a}_i$, e.g., the $(x_i, y_i, 1)$ measurements, have equal noise. This is definitely not the case in the line-fitting example of Figure A.2b (A.40), since the 1 values are noise-free. To mitigate this, we first subtract the mean x and y values from all the measured points

$$\hat{x}_i = x_i - \bar{x} \tag{A.42}$$

$$\hat{y}_i = y_i - \bar{y} \tag{A.43}$$

and then fit the 2D line equation $a(x - \bar{x}) + b(y - \bar{y}) = 0$ by minimizing

$$E_{\text{TLS}-\text{2Dm}} = \sum_i (a\hat{x}_i + b\hat{y}_i)^2. \tag{A.44}$$

[6] Again, be careful with the variable names here. The measurement equation is $\boldsymbol{a}_i = (x_i, y_i, 1)$ and the unknown parameters are $\boldsymbol{x} = (a, b, c)$.

The more general case where each individual measurement component can have different noise level, as is the case in estimating essential and fundamental matrices (Section 7.2), is called the *heteroscedastic* errors-in-variable (HEIV) model and is discussed by Matei and Meer (2006).

A.3 Non-linear least squares

In many vision problems, such as structure from motion, the least squares problem formulated in (A.23) involves functions $\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{p})$ that are *not* linear in the unknown parameters $\boldsymbol{p}$. This problem is known as *non-linear least squares* or *non-linear regression* (Björck 1996; Madsen, Nielsen, and Tingleff 2004; Nocedal and Wright 2006). It is usually solved by iteratively re-linearizing (A.23) around the current estimate of $\boldsymbol{p}$ using the gradient derivative (Jacobian) $\boldsymbol{J} = \partial \boldsymbol{f}/\partial \boldsymbol{p}$ and computing an incremental improvement $\Delta\boldsymbol{p}$.

As shown in Equations (6.13–6.17), this results in

$$E_{\mathrm{NLS}}(\Delta\boldsymbol{p}) = \sum_i \|\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{p}+\Delta\boldsymbol{p}) - \boldsymbol{x}_i'\|^2 \tag{A.45}$$

$$\approx \sum_i \|\boldsymbol{J}(\boldsymbol{x}_i;\boldsymbol{p})\Delta\boldsymbol{p} - \boldsymbol{r}_i\|^2, \tag{A.46}$$

where the Jacobians $\boldsymbol{J}(\boldsymbol{x}_i;\boldsymbol{p})$ and residual vectors $\boldsymbol{r}_i$ play the same role in forming the normal equations as $\boldsymbol{a}_i$ and b_i in (A.28).

Because the above approximation only holds near a local minimum or for small values of $\Delta\boldsymbol{p}$, the update $\boldsymbol{p} \leftarrow \boldsymbol{p} + \Delta\boldsymbol{p}$ may not always decrease the summed square residual error (A.45). One way to mitigate this problem is to take a smaller step,

$$\boldsymbol{p} \leftarrow \boldsymbol{p} + \alpha\Delta\boldsymbol{p}, \quad 0 < \alpha \leq 1. \tag{A.47}$$

A simple way to determine a reasonable value of α is to start with 1 and successively halve the value, which is a simple form of *line search* (Al-Baali and Fletcher. 1986; Björck 1996; Nocedal and Wright 2006).

Another approach to ensuring a downhill step in error is to add a diagonal damping term to the approximate Hessian

$$\boldsymbol{C} = \sum_i \boldsymbol{J}^T(\boldsymbol{x}_i)\boldsymbol{J}(\boldsymbol{x}_i), \tag{A.48}$$

i.e., to solve

$$[\boldsymbol{C} + \lambda\,\mathrm{diag}(\boldsymbol{C})]\Delta\boldsymbol{p} = \boldsymbol{d}, \tag{A.49}$$

where

$$\boldsymbol{d} = \sum_i \boldsymbol{J}^T(\boldsymbol{x}_i)\boldsymbol{r}_i, \tag{A.50}$$

which is called a *damped Gauss–Newton* method. The damping parameter λ is increased if the squared residual is not decreasing as fast as expected, i.e., as predicted by (A.46), and is decreased if the expected decrease is obtained (Madsen, Nielsen, and Tingleff 2004). The combination of the Newton (first-order Taylor series) approximation (A.46) and the adaptive damping parameter λ is commonly known as the Levenberg–Marquardt algorithm (Levenberg 1944; Marquardt 1963) and is an example of more general *trust region methods*, which are discussed in more detail in (Björck 1996; Conn, Gould, and Toint 2000; Madsen, Nielsen, and Tingleff 2004; Nocedal and Wright 2006).

When the initial solution is far away from its quadratic region of convergence around a local minimum, *large residual methods*, e.g., *Newton-type methods*, which add a second-order term to the Taylor series expansion in (A.46), may converge faster. Quasi-Newton methods such as BFGS, which require only gradient evaluations, can also be useful if memory size is an issue. Such techniques are discussed in textbooks and papers on numerical optimization (Toint 1987; Björck 1996; Conn, Gould, and Toint 2000; Nocedal and Wright 2006).

A.4 Direct sparse matrix techniques

Many optimization problems in computer vision, such as bundle adjustment (Szeliski and Kang 1994; Triggs, McLauchlan, Hartley *et al.* 1999; Hartley and Zisserman 2004; Snavely, Seitz, and Szeliski 2008b; Agarwal, Snavely, Simon *et al.* 2009) have Jacobian and (approximate) Hessian matrices that are extremely sparse (Section 7.4.1). For example, Figure 7.9a shows the *bipartite* model typical of structure from motion problems, in which most points are only observed by a subset of the cameras, which results in the sparsity patterns for the Jacobian and Hessian shown in Figure 7.9b–c.

Whenever the Hessian matrix is sparse enough, it is more efficient to use sparse Cholesky factorization instead of regular Cholesky factorization. In such sparse direct techniques, the Hessian matrix $\boldsymbol{C}$ and its associated Cholesky factor $\boldsymbol{R}$ are stored in *compressed form*, in which the amount of storage is proportional to the number of (potentially) non-zero entries (Björck 1996; Davis 2006).[7] Algorithms for computing the non-zero elements in $\boldsymbol{C}$ and $\boldsymbol{R}$ from the sparsity pattern of the Jacobian matrix $\boldsymbol{J}$ are given by Björck (1996, Section 6.4), and algorithms for computing the numerical Cholesky and QR decompositions (once the sparsity pattern has been computed and storage allocated) are discussed by Björck (1996, Section 6.5).

[7] For example, you can store a list of (i, j, c_{ij}) triples. One example of such a scheme is *compressed sparse row (CSR)* storage. An alternative storage method called *skyline*, which stores adjacent vertical spans of non-zero elements (Bathe 2007), is sometimes used in finite element analysis. Banded systems such as snakes (5.3) can store just the non-zero band elements (Björck 1996, Section 6.2) and can be solved in $O(nb^2)$, where n is the number of variables and b is the bandwidth.

A.4.1 Variable reordering

The key to efficiently solving sparse problems using direct (non-iterative) techniques is to determine an efficient *ordering* for the variables, which reduces the amount of *fill-in*, i.e., the number of non-zero entries in $\boldsymbol{R}$ that were zero in the original $\boldsymbol{C}$ matrix. We already saw in Section 7.4.1 how storing the more numerous 3D point parameters before the camera parameters and using the Schur complement (7.56) results in a more efficient algorithm. Similarly, sorting parameters by time in video-based reconstruction problems usually results in lower fill-in. Furthermore, any problem whose adjacency graph (the graph corresponding to the sparsity pattern) is a tree can be solved in linear time with an appropriate reordering of the variables (putting all the children before their parents). All of these are examples of good reordering techniques.

In the general case of unstructured data, there are many heuristics available to find good reorderings (Björck 1996; Davis 2006).[8] For general adjacency (sparsity) graphs, *minimum degree orderings* generally produce good results. For planar graphs, which often arise on image or spline grids (Section 8.3), *nested dissection*, which recursively splits the graph into two equal halves along a *frontier* (or boundary) of small size, generally works well. Such *domain decomposition* (or *multi-frontal*) techniques also enable the use of parallel processing, since independent sub-graphs can be processed in parallel on separate processors (Davis 2008).

The overall set of steps used to perform the direct solution of sparse least squares problems are summarized in Algorithm A.2, which is a modified version of Algorithm 6.6.1 by Björck (1996, Section 6.6)). If a series of related least squares problems is being solved, as is the case in iterative non-linear least squares (Appendix A.3), steps 1–3 can be performed ahead of time and reused for each new invocation with different $\boldsymbol{C}$ and $\boldsymbol{d}$ values. When the problem is block-structured, as is the case in structure from motion where point (structure) variables have dense 3×3 sub-entries in $\boldsymbol{C}$ and cameras have 6×6 (or larger) entries, the cost of performing the reordering computation is small compared to the actual numerical factorization, which can benefit from block-structured matrix operations (Golub and Van Loan 1996). It is also possible to apply sparse reordering and multifrontal techniques to QR factorization (Davis 2008), which may be preferable when the least squares problems are poorly conditioned.

A.5 Iterative techniques

When problems become large, the amount of memory required to store the Hessian matrix $\boldsymbol{C}$ and its factor $\boldsymbol{R}$, and the amount of time it takes to compute the factorization, can become prohibitively large, especially when there are large amounts of fill-in. This is often

[8] Finding the optimal reordering with minimal fill-in is provably NP-hard.

procedure *SparseCholeskySolve*($\boldsymbol{C}, \boldsymbol{d}$):

1. Determine symbolically the structure of $\boldsymbol{C}$, i.e., the adjacency graph.
2. (Optional) Compute a reordering for the variables, taking into account any block structure inherent in the problem.
3. Determine the fill-in pattern for $\boldsymbol{R}$ and allocate the compressed storage for $\boldsymbol{R}$ as well as storage for the permuted right hand side $\hat{\boldsymbol{d}}$.
4. Copy the elements of $\boldsymbol{C}$ and $\boldsymbol{d}$ into $\boldsymbol{R}$ and $\hat{\boldsymbol{d}}$, permuting the values according to the computed ordering.
5. Perform the numerical factorization of $\boldsymbol{R}$ using Algorithm A.1.
6. Solve the factored system (A.33), i.e.,
$$\boldsymbol{R}^T \boldsymbol{z} = \hat{\boldsymbol{d}}, \quad \boldsymbol{R}\boldsymbol{x} = \boldsymbol{z}.$$
7. Return the solution $\boldsymbol{x}$, after undoing the permutation.

Algorithm A.2 Sparse least squares using a sparse Cholesky decomposition of the matrix $\boldsymbol{C}$.

the case with image processing problems defined on pixel grids, since, even with the optimal reordering (nested dissection) the amount of fill can still be large.

A preferable approach to solving such linear systems is to use iterative techniques, which compute a series of estimates that converge to the final solution, e.g., by taking a series of downhill steps in an energy function such as (A.29).

A large number of iterative techniques have been developed over the years, including such well-known algorithms as successive overrelaxation and multi-grid. These are described in specialized textbooks on iterative solution techniques (Axelsson 1996; Saad 2003) as well as in more general books on numerical linear algebra and least squares techniques (Björck 1996; Golub and Van Loan 1996; Trefethen and Bau 1997; Nocedal and Wright 2006; Björck and Dahlquist 2010).

A.5.1 Conjugate gradient

The iterative solution technique that often performs best is conjugate gradient descent, which takes a series of downhill steps that are *conjugate* to each other with respect to the $\boldsymbol{C}$ matrix,

i.e., if the $\boldsymbol{u}$ and $\boldsymbol{v}$ descent directions satisfy $\boldsymbol{u}^T\boldsymbol{C}\boldsymbol{v} = 0$. In practice, conjugate gradient descent outperforms other kinds of gradient descent algorithm because its convergence rate is proportional to the square root of the *condition number* of $\boldsymbol{C}$ instead of the condition number itself.[9] Shewchuk (1994) provides a nice introduction to this topic, with clear intuitive explanations of the reasoning behind the conjugate gradient algorithm and its performance.

Algorithm A.3 describes the conjugate gradient algorithm and its related least squares counterpart, which can be used when the original set of least squares linear equations are available in the form of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ (A.28). While it is easy to convince yourself that the two forms are mathematically equivalent, the least squares form is preferable if rounding errors start to affect the results because of poor conditioning. It may also be preferable if, due to the sparsity structure of $\boldsymbol{A}$, multiplies with the original $\boldsymbol{A}$ matrix are faster or more space efficient than multiplies with $\boldsymbol{C}$.

The conjugate gradient algorithm starts by computing the current residual $\boldsymbol{r}_0 = \boldsymbol{d} - \boldsymbol{C}\boldsymbol{x}_0$, which is the direction of steepest descent of the energy function (A.28). It sets the original descent direction $\boldsymbol{p}_0 = \boldsymbol{r}_0$. Next, it multiplies the descent direction by the quadratic form (Hessian) matrix $\boldsymbol{C}$ and combines this with the residual to estimate the optimal step size α_k. The solution vector $\boldsymbol{x}_k$ and the residual vector $\boldsymbol{r}_k$ are then updated using this step size. (Notice how the least squares variant of the conjugate gradient algorithm splits the multiplication by the $\boldsymbol{C} = \boldsymbol{A}^T\boldsymbol{A}$ matrix across steps 4 and 8.) Finally, a new search direction is calculated by first computing a factor β as the ratio of current to previous residual magnitudes. The new search direction $\boldsymbol{p}_{k+1}$ is then set to the residual plus β times the old search direction $\boldsymbol{p}_k$, which keeps the directions conjugate with respect to $\boldsymbol{C}$.

It turns out that conjugate gradient descent can also be directly applied to non-quadratic energy functions, e.g., those arising from non-linear least squares (Appendix A.3). Instead of explicitly forming a local quadratic approximation $\boldsymbol{C}$ and then computing residuals $\boldsymbol{r}_k$, non-linear conjugate gradient descent computes the gradient of the energy function E (A.45) directly inside each iteration and uses it to set the search direction (Nocedal and Wright 2006). Since the quadratic approximation to the energy function may not exist or may be inaccurate, line search is often used to determine the step size α_k. Furthermore, to compensate for errors in finding the true function minimum, alternative formulas for β_{k+1} such as Polak–Ribière,

$$\beta_{k+1} = \frac{\nabla E(\boldsymbol{x}_{k+1})[\nabla E(\boldsymbol{x}_{k+1}) - \nabla E(\boldsymbol{x}_k)]}{\|\nabla E(\boldsymbol{x}_k)\|^2} \tag{A.51}$$

are often used (Nocedal and Wright 2006).

[9] The condition number $\kappa(\boldsymbol{C})$ is the ratio of the largest and smallest eigenvalues of $\boldsymbol{C}$. The actual convergence rate depends on the clustering of the eigenvalues, as discussed in the references cited in this section.

ConjugateGradient$(\boldsymbol{C}, \boldsymbol{d}, \boldsymbol{x}_0)$

1. $\boldsymbol{r}_0 = \boldsymbol{d} - \boldsymbol{C}\boldsymbol{x}_0$
2. $\boldsymbol{p}_0 = \boldsymbol{r}_0$
3. **for** $k = 0 \ldots$
4. $\quad \boldsymbol{w}_k = \boldsymbol{C}\boldsymbol{p}_k$
5. $\quad \alpha_k = \|\boldsymbol{r}_k\|^2 / (\boldsymbol{p}_k \cdot \boldsymbol{w}_k)$
6. $\quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
7. $\quad \boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k \boldsymbol{w}_k$
8.
9. $\quad \beta_{k+1} = \|\boldsymbol{r}_{k+1}\|^2 / \|\boldsymbol{r}_k\|^2$
10. $\quad \boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$

ConjugateGradientLS$(\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}_0)$

1. $\boldsymbol{q}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0, \quad \boldsymbol{r}_0 = \boldsymbol{A}^T \boldsymbol{q}_0$
2. $\boldsymbol{p}_0 = \boldsymbol{r}_0$
3. **for** $k = 0 \ldots$
4. $\quad \boldsymbol{v}_k = \boldsymbol{A}\boldsymbol{p}_k$
5. $\quad \alpha_k = \|\boldsymbol{r}_k\|^2 / \|\boldsymbol{v}_k\|^2$
6. $\quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$
7. $\quad \boldsymbol{q}_{k+1} = \boldsymbol{q}_k - \alpha_k \boldsymbol{v}_k$
8. $\quad \boldsymbol{r}_{k+1} = \boldsymbol{A}^T \boldsymbol{q}_{k+1}$
9. $\quad \beta_{k+1} = \|\boldsymbol{r}_{k+1}\|^2 / \|\boldsymbol{r}_k\|^2$
10. $\quad \boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$

Algorithm A.3 Conjugate gradient and conjugate gradient least squares algorithms. The algorithm is described in more detail in the text, but in brief, they choose descent directions $\boldsymbol{p}_k$ that are conjugate to each other with respect to $\boldsymbol{C}$ by computing a factor β by which to discount the previous search direction $\boldsymbol{p}_{k-1}$. They then find the optimal step size α and take a downhill step by an amount $\alpha_k \boldsymbol{p}_k$.

A.5.2 Preconditioning

As we mentioned previously, the rate of convergence of the conjugate gradient algorithm is governed in large part by the condition number $\kappa(\boldsymbol{C})$. Its effectiveness can therefore be increased dramatically by reducing this number, e.g., by rescaling elements in $\boldsymbol{x}$, which corresponds to rescaling rows and columns in $\boldsymbol{C}$.

In general, preconditioning is usually thought of as a change of basis from the vector $\boldsymbol{x}$ to a new vector

$$\hat{x} = \boldsymbol{S}\boldsymbol{x}. \tag{A.52}$$

The corresponding linear system being solved then becomes

$$\boldsymbol{A}\boldsymbol{S}^{-1}\hat{x} = \boldsymbol{S}^{-1}\boldsymbol{b} \quad \text{or} \quad \hat{\boldsymbol{A}}\hat{x} = \hat{\boldsymbol{b}}, \tag{A.53}$$

with a corresponding least squares energy (A.29) of the form

$$E_{\mathrm{PLS}} = \hat{x}^T(\boldsymbol{S}^{-T}\boldsymbol{C}\boldsymbol{S}^{-1})\hat{x} - 2\hat{x}^T(\boldsymbol{S}^{-T}\boldsymbol{d}) + \|\hat{\boldsymbol{b}}\|^2. \tag{A.54}$$

The actual preconditioned matrix $\hat{\boldsymbol{C}} = \boldsymbol{S}^{-T}\boldsymbol{C}\boldsymbol{S}^{-1}$ is usually not explicitly computed. Instead, Algorithm A.3 is extended to insert $\boldsymbol{S}^{-T}$ and $\boldsymbol{S}^T$ operations at the appropriate places (Björck 1996; Golub and Van Loan 1996; Trefethen and Bau 1997; Saad 2003; Nocedal and Wright 2006).

A good preconditioner $\boldsymbol{S}$ is easy and cheap to compute, but is also a decent approximation to a square root of $\boldsymbol{C}$, so that $\kappa(\boldsymbol{S}^{-T}\boldsymbol{C}\boldsymbol{S}^{-1})$ is closer to 1. The simplest such choice is the square root of the diagonal matrix $\boldsymbol{S} = \boldsymbol{D}^{1/2}$, with $\boldsymbol{D} = \mathrm{diag}(\boldsymbol{C})$. This has the advantage that any scalar change in variables (e.g., using radians instead of degrees for angular measurements) has no effect on the range of convergence of the iterative technique. For problems that are naturally block-structured, e.g., for structure from motion, where 3D point positions or 6D camera poses are being estimated, a block diagonal preconditioner is often a good choice.

A wide variety of more sophisticated preconditioners have been developed over the years (Björck 1996; Golub and Van Loan 1996; Trefethen and Bau 1997; Saad 2003; Nocedal and Wright 2006), many of which can be directly applied to problems in computer vision (Byröd and øAström 2009; Jeong, Nistér, Steedly *et al.* 2010; Agarwal, Snavely, Seitz *et al.* 2010). Some of these are based on an *incomplete Cholesky* factorization of $\boldsymbol{C}$, i.e., one in which the amount of fill-in in $\boldsymbol{R}$ is strictly limited, e.g., to just the original non-zero elements in $\boldsymbol{C}$.[10] Other preconditioners are based on a sparsified, e.g., tree-based or clustered, approximation to $\boldsymbol{C}$ (Koutis 2007; Koutis and Miller 2008; Grady 2008; Koutis, Miller, and Tolliver 2009), since these are known to have efficient inversion properties.

For grid-based image-processing applications, *parallel* or *hierarchical* preconditioners often perform extremely well (Yserentant 1986; Szeliski 1990b; Pentland 1994; Saad 2003; Szeliski 2006b). These approaches use a change of basis transformation $\boldsymbol{S}$ that resembles the pyramidal or wavelet representations discussed in Section 3.5, and are hence amenable to parallel and GPU-based implementations. Coarser elements in the new representation quickly converge to the low-frequency components in the solution, while finer-level elements encode the higher-frequency components. Some of the relationships between hierarchical preconditioners, incomplete Cholesky factorization, and multigrid techniques are explored by Saad (2003) and Szeliski (2006b).

[10] If a complete Cholesky factorization $\boldsymbol{C} = \boldsymbol{R}^T\boldsymbol{R}$ is used, we get $\hat{\boldsymbol{C}} = \boldsymbol{R}^{-T}\boldsymbol{C}\boldsymbol{R}^{-1} = \boldsymbol{I}$ and all iterative algorithms converge in a single step, thereby obviating the need to use them, but the complete factorization is often too expensive. Note that incomplete factorization can also benefit from reordering.

A.5.3 Multigrid

One other class of iterative techniques widely used in computer vision is *multigrid* techniques (Briggs, Henson, and McCormick 2000; Trottenberg, Oosterlee, and Schuller 2000), which have been applied to problems such as surface interpolation (Terzopoulos 1986a), optical flow (Terzopoulos 1986a; Bruhn, Weickert, Kohlberger *et al.* 2006), high dynamic range tone mapping (Fattal, Lischinski, and Werman 2002), colorization (Levin, Lischinski, and Weiss 2004), natural image matting (Levin, Lischinski, and Weiss 2008), and segmentation (Grady 2008).

The main idea behind multigrid is to form coarser (lower-resolution) versions of the problems and use them to compute the low-frequency components of the solution. However, unlike simple coarse-to-fine techniques, which use the coarse solutions to initialize the fine solution, multigrid techniques only *correct* the low-frequency component of the current solution and use multiple rounds of coarsening and refinement (in what are often called "V" and "W" patterns of motion across the pyramid) to obtain rapid convergence.

On certain simple homogeneous problems (such as solving Poisson equations), multigrid techniques can achieve optimal performance, i.e., computation times linear in the number of variables. However, for more inhomogeneous problems or problems on irregular grids, variants on these techniques, such as *algebraic multigrid* (AMG) approaches, which look at the structure of $\boldsymbol{C}$ to derive coarse level problems, may be preferable. Saad (2003) has a nice discussion of the relationship between multigrid and parallel preconditioners and on the relative merits of using multigrid or conjugate gradient approaches.

Appendix B

Bayesian modeling and inference

B.1 Estimation theory . . . 757
B.1.1 Likelihood for multivariate Gaussian noise . . . 757
B.2 Maximum likelihood estimation and least squares . . . 759
B.3 Robust statistics . . . 760
B.4 Prior models and Bayesian inference . . . 762
B.5 Markov random fields . . . 763
B.5.1 Gradient descent and simulated annealing . . . 765
B.5.2 Dynamic programming . . . 766
B.5.3 Belief propagation . . . 768
B.5.4 Graph cuts . . . 770
B.5.5 Linear programming . . . 773
B.6 Uncertainty estimation (error analysis) . . . 775