SBA: A Software Package for Generic Sparse Bundle Adjustment

MANOLIS I. A. LOURAKIS and ANTONIS A. ARGYROS
Foundation for Research and Technology—Hellas

Bundle adjustment constitutes a large, nonlinear least-squares problem that is often solved as the last step of feature-based structure and motion estimation computer vision algorithms to obtain optimal estimates. Due to the very large number of parameters involved, a general purpose least-squares algorithm incurs high computational and memory storage costs when applied to bundle adjustment. Fortunately, the lack of interaction among certain subgroups of parameters results in the corresponding Jacobian being sparse, a fact that can be exploited to achieve considerable computational savings. This article presents `sba`, a publicly available C/C++ software package for realizing generic bundle adjustment with high efficiency and flexibility regarding parameterization.

Categories and Subject Descriptors: I.4.8 [**Image Processing and Computer Vision**]: Scene Analysis—*Motion, shape, stereo, time-varying imagery, tracking*; G.4 [**Mathematical Software**]: *Algorithm design and analysis, efficiency*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Linear systems (direct and iterative methods), sparse, structural, and very large systems (direct and iterative methods)*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Unconstrained optimization, nonlinear least squares, Levenberg-Marquardt, sparse Jacobian, bundle adjustment, structure and motion estimation, multiple-view geometry, engineering applications

ACM Reference Format:

Lourakis, M. I. A. and Argyros, A. A. 2009. SBA: A software package for generic sparse bundle adjustment. ACM Trans. Math. Softw. 36, 1, Article 2 (March 2009), 30 pages. DOI = 10.1145/1486525.1486527 http://doi.acm.org/10.1145/1486525.1486527

1. INTRODUCTION

Three-dimensional (3D) reconstruction is an engineering problem whose solution is called for by a wide spectrum of computer vision tasks. Generally

The financial support of the European Union through projects IST-2001-34545 Lifeplus, COOP-CT-2005-017405 RECOVER, and FP6-507752 NoE MUSCLE is acknowledged.
Authors' addresses: Computational Vision and Robotics Laboratory, Institute of Computer Science, Foundation for Research and Technology-Hellas (FORTH), N. Plastira 100, Vassilika Vouton, 700 13, Heraklion, Crete, Greece; email: `{lourakis,argyros}@ics.forth.gr`.

speaking, 3D reconstruction can be defined as the problem of using 2D measurements arising from a set of images depicting the same scene from different viewpoints, aiming to derive information related to the 3D scene geometry as well as the relative motion and the optical characteristics of the camera(s) employed to acquire these images. Bundle adjustment (BA) is almost invariably used as the last step of every feature-based 3D reconstruction algorithm; see, for example, Hartley [1993]; Szeliski and Kang [1994]; Beardsley et al. [1996]; Fitzgibbon and Zisserman [1998]; Shum et al. [1999]; Zhang and Shan [2003]; Pollefeys et al. [2004]; Lourakis and Argyros [2005a]; Snavely et al. [2006] for a few representative approaches. BA amounts to an optimization problem on the 3D structure and viewing parameters (i.e., camera pose and possibly intrinsic calibration and radial distortion), to obtain a reconstruction which is optimal under certain assumptions regarding the noise pertaining to the observed image features [Triggs et al. 1999]: if the image error is zero-mean Gaussian, then BA is the maximum likelihood estimator. Its name refers to the "bundles" of light rays originating from each 3D feature and converging on each camera's optical center, which are adjusted optimally with respect to both the structure and viewing parameters. BA was originally conceived in the field of photogrammetry during 1950s [Brown 1958; Slama 1980] and has increasingly been used by computer vision researchers during recent years. An excellent overview of its application to vision-based reconstruction is given in Triggs et al. [1999]. Apart from computer vision, BA can also find useful applications in areas such as robotics, image-based computer graphics, digital photogrammetry, aerial mapping, industrial metrology, surveying, and geodesy, etc.

BA boils down to minimizing the reprojection error between the observed and predicted image points, which is expressed as the sum of squares of a large number of nonlinear, real-valued functions. Thus, the minimization is achieved using nonlinear least-squares algorithms [Dennis 1977], from which Levenberg-Marquardt (LM) has proven to be of the most successful due to its ease of implementation and its use of an effective damping strategy that lends it the ability to converge quickly from a wide range of initial guesses [Hiebert 1981]. By iteratively linearizing the function to be minimized in the neighborhood of the current estimate, the LM algorithm involves the solution of linear systems known as the *normal equations*. Considering that the normal equations are solved repeatedly in the course of the LM algorithm and that each computation of the solution to a dense linear system has complexity $O(N^3)$ in the number of unknown parameters, it is clear that general-purpose LM codes such as, for example, MINPACK's LMDER routine [Moré et al. 1980], are computationally very demanding when employed to minimize functions depending on a large number of parameters. This observation remains true even if nonlinear least-squares algorithms other than LM are employed (e.g., the NL2SOL algorithm [Dennis et al. 1981] as implemented by PORT3's DN2G[1] routine [Gay 1990]). The situation is further complicated by the fact that the size of the

[1]By keeping the terms due to the second derivative in the Hessian of $\epsilon^T \epsilon$ (see Section 3), routine DN2G actually implements a more general strategy compared to that of the classical Levenberg-Marquardt and is often more appropriate for large residual problems.

Jacobian of the objective function also increases with the number of parameters. Thus, when performing operations involving such a large Jacobian, special care has to be taken to avoid thrashing, that is, instead of performing useful computations, wasting most CPU cycles for writing virtual memory pages out to disk and reading them back in. Fortunately, when solving minimization problems arising in BA, the normal equations matrix has a sparse block structure owing to the lack of interaction among parameters for different 3D points and cameras. Therefore, considerable computational benefits can be gained by developing a tailored, sparse variant of the LM algorithm which explicitly takes advantage of the normal equations zeroes pattern by avoiding storing and operating on zero elements.

The contribution of this article is threefold. First, it describes a strategy for efficiently dealing with the problem of BA. Second, it materializes this strategy through the design and implementation of a software package for BA. Third, it demonstrates experimentally the impact that the exploitation of the problem's structure has on computational performance. The practical outcome of this work is `sba`, a generic sparse BA package implemented in ANSI C. C was preferred over higher-level programming environments such as MATLAB owing to its far superior execution performance and its wide availability in a diverse range of computer systems. Nevertheless, `sba` includes a MEX-file external interface that enables its use directly from MATLAB. `sba` is also usable from C++ and is generic in the sense that it grants the user full control over the choice of coordinate systems, parameters and functional relations describing cameras, 3D structure and image projections. Therefore, it can support a wide range of manifestations/parameterizations of the multiple view reconstruction problem such as arbitrary projective, affine or omnidirectional cameras, partially or fully intrinsically calibrated cameras, exterior orientation (i.e., pose) estimation from fixed 3D points, 3D reconstruction from extrinsically calibrated images, refinement of intrinsic calibration parameters, etc. The `sba` package can be downloaded in source form from `http://www.ics.forth.gr/~lourakis/sba` and is distributed under the terms of the GNU General Public License.[2]

The rest of the article is organized as follows. Section 2 provides a short introduction to the geometry of perspective projection, helping the reader to gain some insight into the BA problem. Section 3 briefly explains the conventional, dense LM algorithm for solving nonlinear least-squares minimization problems. Section 4 develops a sparse BA algorithm by adapting the LM to exploit the sparse block structure of the normal equations. Technical details regarding the implementation and use of the `sba` package are given in Section 5. Experimental results from the use of `sba` on real problems are presented in Section 6 and the article is concluded with a brief discussion in Section 7.

2. CAMERA IMAGING GEOMETRY

In abstract terms, a camera is a device that performs central projection of the 3D world onto a 2D image plane. This is illustrated in Figure 1. For the

[2]See `http://www.gnu.org/copyleft/gpl.html`.

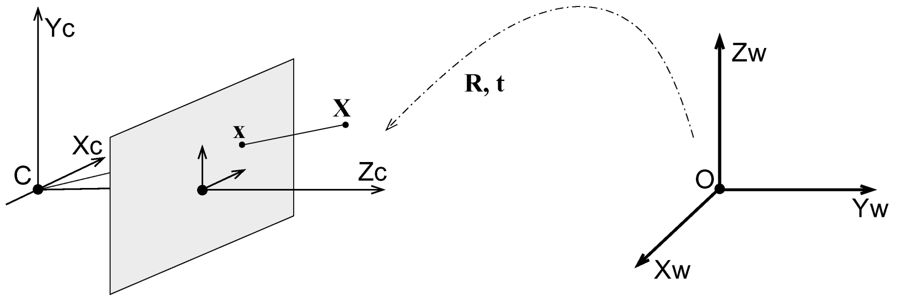

Fig. 1. Perspective projection of a 3D point **X** on an image point **x**. Note that the center of projection C is collinear with **x** and **X**. The coordinate system C Xc Yc Zc attached to the camera is related to the world coordinate system O Xw Yw Zw through a rotation **R** followed by a translation **t**.

purposes of this work, the objects of interest in the world are simply sets of 3D points. To facilitate the representation of central projection using familiar linear algebra operations, vision researchers often employ *projective geometry*, which involves vectors and matrices defined up to scale [Mundy and Zisserman 1992; Semple and Kneebone 1952]. Thus, assuming that both world and image points are represented by homogeneous vectors, central projection is expressed as a linear mapping between their homogeneous coordinates. This mapping is written compactly as

$$\lambda \mathbf{x} = \mathbf{M}\mathbf{X}, \tag{1}$$

where $\mathbf{X}$ is a 4×1 vector representing a 3D point projecting on an image point represented by a 3×1 vector $\mathbf{x}$ and λ is an arbitrary scale factor. Matrix $\mathbf{M}$, also known as the *projection* or *camera matrix*, is an arbitrary homogeneous 3×4 matrix having rank 3 and depending on 11 parameters [Hartley and Zisserman 2000].

The camera model defined by Equation (1) is known as *projective camera* and encompasses several interesting imaging geometries such as weak perspective and affine. A particularly interesting specialization of the general projective camera is the *finite projective* or *Euclidean* camera, for which the left 3×3 submatrix of $\mathbf{M}$ is nonsingular. In this case, $\mathbf{M}$ can be further decomposed as[3] $\mathbf{K}\,[\mathbf{R} \mid \mathbf{t}]$, where $\mathbf{K}$ is the 3×3, upper triangular *intrinsic calibration* matrix which encodes the camera's optical properties (i.e., focal length, aspect ratio, and principal point), $\mathbf{R}$ is an orthogonal 3×3 matrix and $\mathbf{t}$ a 3×1 vector. $\mathbf{R}$ and $\mathbf{t}$ are collectively referred to as the camera's *extrinsic orientation* and correspond, respectively, to the rotation and translation that make up the rigid transformation from the world to the camera coordinate frame (see also Figure 1). With the above definitions, the problem of multiview 3D reconstruction can now be defined more precisely as follows. Assume that n 3D points are observed in m images and denote by $\mathbf{x}_{ij}$ the projection of the ith point on the jth image. Multiview 3D reconstruction amounts to finding the m camera matrices $\mathbf{M}_j$ and the n 3D points $\mathbf{X}_i$ such that each $\mathbf{x}_{ij}$ is closely approximated by

[3]We use the notation $[\mathbf{M} \mid \mathbf{v}]$ to denote the matrix that results by augmenting matrix $\mathbf{M}$ with a new column equal to vector $\mathbf{v}$.

the predicted projection $\mathbf{M}_j \mathbf{X}_i$. For obvious reasons, the reconstruction problem is also known as the *structure and motion* estimation problem. Several algorithms have been proposed for obtaining preliminary, sometimes approximate solutions to the multiple view reconstruction problem. Typically, such solutions serve as starting points for bootstrapping more accurate refinements that are based on BA. At this point, it should be pointed out that in practice, the $\mathbf{x}_{ij}$ are the outcome of a measurement process involving signal processing algorithms (e.g., Shi and Tomasi [1994]; Lowe [2004]). Therefore, they are subject to noise, that is, measurement errors which come in two forms: *localization* errors that arise when image projections are not detected at their correct image locations and *matching* errors that are the result of erroneous associations of image projections across images. For this reason, every reconstruction algorithm should anticipate and be robust to the presence of noise. For the remainder of the article, it will be assumed that while image projections are subject to localization errors, erroneous matches, that is, gross outliers, have been detected and eliminated using geometry-based techniques such as those described in Zhang et al. [1995].

3. THE LEVENBERG-MARQUARDT ALGORITHM

The LM algorithm, originally suggested by Levenberg [Levenberg 1944] and later by Marquardt [Marquardt 1963], is an iterative technique that finds a local minimum of a multivariate function that is expressed as the sum of squares of nonlinear real-valued functions. It has become a standard technique for nonlinear least-squares problems, widely adopted in various disciplines for dealing with data-fitting applications. LM can be thought of as a combination of steepest descent and the Gauss-Newton method. When the current solution is far from a local minimum, the algorithm behaves like a steepest descent method: slow, but guaranteed to converge. When the current solution is close to a local minimum, it becomes a Gauss-Newton method and exhibits fast convergence. For the sake of completeness, a short description of the LM algorithm based on the material in Madsen et al. [2004] is supplied next. Note, however, that a detailed analysis of the LM algorithm is beyond the scope of this article and the interested reader is referred to Nocedal and Wright [1999], Kelley [1999], and Madsen et al. [2004] for more extensive treatments.

In the following, vectors and arrays appear in boldface and ^{T} is used to denote transposition. Also, $||.||$ and $||.||_\infty$, respectively, denote the L2 and infinity norms. Let f be an assumed functional relation which maps a *parameter vector* $\mathbf{p} \in \mathcal{R}^m$ to an estimated *measurement vector* $\hat{\mathbf{x}} = f(\mathbf{p})$, $\hat{\mathbf{x}} \in \mathcal{R}^n$. An initial parameter estimate $\mathbf{p}_0$ and a measured vector $\mathbf{x}$ are provided and it is desired to find the vector $\mathbf{p}^+$ that best satisfies the functional relation f locally, that is, minimizes the squared distance $\epsilon^T \epsilon$ with $\epsilon = \mathbf{x} - \hat{\mathbf{x}}$ for all $\mathbf{p}$ within a sphere having a certain, small radius. The basis of the LM algorithm is an affine approximation to f in the neighborhood of $\mathbf{p}$. For a small $||\delta_\mathbf{p}||$, f is approximated by (see Dennis and Schnabel [1996], p. 75)

$$f(\mathbf{p} + \delta_\mathbf{p}) \approx f(\mathbf{p}) + \mathbf{J}\delta_\mathbf{p}, \tag{2}$$

where $\mathbf{J}$ is the Jacobian of f. Like all nonlinear optimization methods, LM is iterative: initiated at the starting point $\mathbf{p}_0$, it produces a series of vectors $\mathbf{p}_1, \mathbf{p}_2, \ldots,$ that converge toward a local minimizer $\mathbf{p}^+$ for f. Hence, at each iteration, it is required to find the step $\delta_\mathbf{p}$ that minimizes the quantity $||\mathbf{x} - f(\mathbf{p} + \delta_\mathbf{p})|| \approx ||\mathbf{x} - f(\mathbf{p}) - \mathbf{J}\delta_\mathbf{p}|| = ||\epsilon - \mathbf{J}\delta_\mathbf{p}||$. The sought $\delta_\mathbf{p}$ is thus the solution to a linear least-squares problem: the minimum is attained when $\mathbf{J}\delta_\mathbf{p} - \epsilon$ is orthogonal to the column space of $\mathbf{J}$. This leads to $\mathbf{J}^T(\mathbf{J}\delta_\mathbf{p} - \epsilon) = \mathbf{0}$, which yields $\delta_\mathbf{p}$ as the solution of the so-called *normal equations* [Golub and van Loan 1996]:

$$\mathbf{J}^T\mathbf{J}\delta_\mathbf{p} = \mathbf{J}^T\epsilon. \tag{3}$$

Matrix $\mathbf{J}^T\mathbf{J}$ in the above equation is the first order approximation to the Hessian of $\frac{1}{2}\epsilon^T\epsilon$ [Nocedal and Wright 1999] and $\delta_\mathbf{p}$ is the *Gauss-Newton* step. Note also that $\mathbf{J}^T\epsilon$ corresponds to the steepest descent direction since the gradient of $\frac{1}{2}\epsilon^T\epsilon$ is $-\mathbf{J}^T\epsilon$. The LM method actually solves a slight variation of Equation (3), known as the *augmented normal equations*

$$\mathbf{N}\delta_\mathbf{p} = \mathbf{J}^T\epsilon, \text{ with } \mathbf{N} \equiv \mathbf{J}^T\mathbf{J} + \mu\mathbf{I}, \ \mu > 0. \tag{4}$$

The strategy of altering the diagonal elements of $\mathbf{J}^T\mathbf{J}$ is called *damping* and μ is referred to as the *damping term*. If the updated parameter vector $\mathbf{p} + \delta_\mathbf{p}$ with $\delta_\mathbf{p}$ computed from Equation (4) leads to a reduction in the error $\epsilon^T\epsilon$, the update is accepted and the process repeats with a decreased damping term. Otherwise, the damping term is increased, the augmented normal equations are solved again, and the process iterates until a value of $\delta_\mathbf{p}$ that decreases the error is found. The process of repeatedly solving Equation (4) for different values of the damping term until an acceptable update to the parameter vector is found corresponds to one iteration of the LM algorithm.

In LM, the damping term is adjusted at each iteration to assure a reduction in the error. If the damping is set to a large value, matrix $\mathbf{N}$ in Equation (4) is nearly diagonal and the LM update step $\delta_\mathbf{p}$ is skewed toward the steepest descent direction $\mathbf{J}^T\epsilon$. Moreover, the magnitude of $\delta_\mathbf{p}$ is reduced in this case, ensuring that excessively large Gauss-Newton steps are not taken. Damping also handles situations where the Jacobian is rank deficient and $\mathbf{J}^T\mathbf{J}$ is therefore singular [Dennis and Schnabel 1996; Lampton 1997]. The damping term can be chosen so that matrix $\mathbf{N}$ in Equation (4) is safely nonsingular and, therefore, positive definite, thus ensuring that the $\delta_\mathbf{p}$ computed from it is a descent direction. If the damping is small, the LM step approximates the Newton minimizer of the local quadratic model $m(\delta_\mathbf{p}) = \frac{1}{2}\epsilon^T\epsilon - (\mathbf{J}^T\epsilon)^T\delta_\mathbf{p} + \frac{1}{2}\delta_\mathbf{p}^T\mathbf{J}^T\mathbf{J}\delta_\mathbf{p}$ of $\frac{1}{2}\epsilon^T\epsilon$ about $\mathbf{p}$ (cf. Equation (3)), which is exact in the case of a fully linear problem [Kelley 1999]. LM is adaptive because it controls its own damping: it raises the damping if a step fails to reduce $\epsilon^T\epsilon$; otherwise it reduces the damping. By doing so, LM is capable of alternating between a slow descent approach when being far from the minimum and a fast convergence when being at the minimum's neighborhood [Lampton 1997]. An efficient updating strategy for the damping term that is also used in this work has been described in Nielsen [1999]. The LM algorithm terminates when any of the following conditions is met:

—the magnitude of the gradient drops below a threshold ε_1;
—the relative magnitude of $\delta_\mathbf{p}$ drops below a threshold involving a parameter ε_2;

```
Input: A vector function $f : \mathcal{R}^m \rightarrow \mathcal{R}^n$ with $n \geq m$, a measurement vector $\mathbf{x} \in \mathcal{R}^n$ and an
initial parameters estimate $\mathbf{p}_0 \in \mathcal{R}^m$.
Output: A vector $\mathbf{p}^+ \in \mathcal{R}^m$ minimizing $||\mathbf{x} - f(\mathbf{p})||^2$.
Algorithm:
$k := 0$; $\nu := 2$; $\mathbf{p} := \mathbf{p}_0$;
$\mathbf{A} := \mathbf{J}^T\mathbf{J}$; $\epsilon_\mathbf{p} := \mathbf{x} - f(\mathbf{p})$; $\mathbf{g} := \mathbf{J}^T\epsilon_\mathbf{p}$;
stop:=$(||\mathbf{g}||_\infty \leq \varepsilon_1)$; $\mu := \tau * \max_{i=1,\ldots,m}(A_{ii})$;
while (not stop) and $(k < k_{max})$
    $k := k + 1$;
    repeat
        $\boxed{\text{Solve } (\mathbf{A} + \mu\mathbf{I})\delta_\mathbf{p} = \mathbf{g};}$
        if $(||\delta_\mathbf{p}|| \leq \varepsilon_2(||\mathbf{p}|| + \varepsilon_2))$
            stop:=true;
        else
            $\mathbf{p}_{new} := \mathbf{p} + \delta_\mathbf{p}$;
            $\rho := (||\epsilon_\mathbf{p}||^2 - ||\mathbf{x} - f(\mathbf{p}_{new})||^2)/(\delta_\mathbf{p}^T(\mu\delta_\mathbf{p} + \mathbf{g}))$;
            if $\rho > 0$
                stop:=$(||\epsilon_\mathbf{p}|| - ||\mathbf{x} - f(\mathbf{p}_{new})|| < \varepsilon_4||\epsilon_\mathbf{p}||)$;
                $\mathbf{p} = \mathbf{p}_{new}$;
                $\mathbf{A} := \mathbf{J}^T\mathbf{J}$; $\epsilon_\mathbf{p} := \mathbf{x} - f(\mathbf{p})$; $\mathbf{g} := \mathbf{J}^T\epsilon_\mathbf{p}$;
                stop:=(stop) or $(||\mathbf{g}||_\infty \leq \varepsilon_1)$;
                $\mu := \mu * \max(\frac{1}{3}, 1 - (2\rho - 1)^3)$; $\nu := 2$;
            else
                $\mu := \mu * \nu$; $\nu := 2 * \nu$;
            endif
        endif
    until $(\rho > 0)$ or (stop)
    stop:=$(||\epsilon_\mathbf{p}|| \leq \varepsilon_3)$;
endwhile
$\mathbf{p}^+ := \mathbf{p}$;
```

Fig. 2. Pseudocode for the Levenberg-Marquardt nonlinear least-squares algorithm; see text for details. ρ is the *gain ratio*, defined by the ratio of the actual reduction in the error $||\epsilon_\mathbf{p}||^2$ that corresponds to a step $\delta_\mathbf{p}$ and the reduction predicted for $\delta_\mathbf{p}$ by the linear model of Equation (2). The sign of ρ determines whether $\delta_\mathbf{p}$ is accepted or not. Furthermore, in the case of accepted steps, the value of ρ controls the reduction in the damping term. The reason for enclosing a statement in a rectangular box will be explained in Section 4.

—the magnitude of the residual ϵ drops below a threshold ε_3;

—the relative reduction in the magnitude of the residual ϵ drops below threshold ε_4;

—a maximum number of iterations k_{max} is reached.

If a covariance matrix $\Sigma_\mathbf{x}$ describing the uncertainty of the measured vector $\mathbf{x}$ is available, it can be incorporated into the LM algorithm by minimizing the squared $\Sigma_\mathbf{x}^{-1}$-norm $\epsilon^T\Sigma_\mathbf{x}^{-1}\epsilon$ instead of the Euclidean norm $\epsilon^T\epsilon$. Accordingly, the minimum is found by solving a weighted least-squares problem defined by the augmented *weighted normal equations*

$$(\mathbf{J}^T\Sigma_\mathbf{x}^{-1}\mathbf{J} + \mu\mathbf{I})\delta_\mathbf{p} = \mathbf{J}^T\Sigma_\mathbf{x}^{-1}\epsilon. \tag{5}$$

Except from substituting the L2 norm of the error $||\epsilon||$ with its $\Sigma_\mathbf{x}^{-1}$-norm, the rest of the algorithm remains unchanged. It is noted that the initial damping factor is chosen equal to the product of a parameter τ with the maximum element of $\mathbf{J}^T\mathbf{J}$ in the main diagonal. The complete LM algorithm is shown in pseudocode in Figure 2; more details regarding it can be found in Madsen

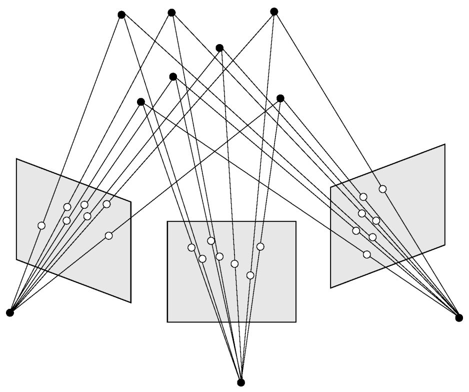

Fig. 3. Schematic illustration of $n = 7$ points projecting on $m = 3$ images.

et al. [2004]. Typical values for the user-defined parameters are $\tau = 10^{-3}$, $\varepsilon_1 = \varepsilon_2 = \varepsilon_3 = 10^{-12}$, $\varepsilon_4 = 0$, $k_{max} = 100$.

At this point, it should be mentioned that, rather than directly controlling the damping parameter μ in Equation (4), modern implementations of the Levenberg-Marquardt algorithm such as Moré [1977] and Moré et al. [1980] seek a nearly exact solution for μ using Newton's root finding algorithm in a trust-region framework [Moré and Sorensen 1983; Conn et al. 2000]. This approach, however, requires expensive repetitive Cholesky factorizations of the augmented approximate Hessian and, therefore, is not well suited to solving large-scale problems such as those arising in the context of BA.

4. SPARSE BUNDLE ADJUSTMENT

This section shows how a sparse variant of the LM algorithm presented in Section 3 can be developed to deal efficiently with the problem of bundle adjustment. The developments that follow are along the lines of the presentation regarding sparse bundle adjustment in Appendix 4 of Hartley and Zisserman [2000]. As illustrated in Figure 3, assume that n 3D points are seen in m views and let $\mathbf{x}_{ij}$ be the (generally two-dimensional) projection of the ith point on image j. Bundle adjustment is equivalent to jointly refining a set of initial camera and structure parameter estimates for finding the set of parameters that most accurately predict the locations of the observed n points in the set of the m available images. More formally, assume that each camera j is parameterized by a vector $\mathbf{a}_j$ and each 3D point i by a vector $\mathbf{b}_i$. For notational simplicity, it is also assumed that all points are visible in all images. This assumption, however, is not necessary and, as will soon be made clear, points may in general be visible in any subset of the m views. BA minimizes the *reprojection error* with respect to all 3D point and camera parameters, specifically

$$\min_{\mathbf{a}_j, \mathbf{b}_i} \sum_{i=1}^{n} \sum_{j=1}^{m} d(\mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i),\ \mathbf{x}_{ij})^2, \tag{6}$$

where $\mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i)$ is the predicted projection of point i on image j and $d(\mathbf{x}, \mathbf{y})$ denotes the Euclidean distance between the image points represented by the inhomogeneous vectors $\mathbf{x}$ and $\mathbf{y}$. It is clear from (6) that BA is by definition tolerant to missing image projections and, in contrast to algebraic approaches for multiview reconstruction, minimizes a physically meaningful criterion. Observe that, through $\mathbf{Q}()$, the definition in (6) is general enough to accommodate any camera and structure parameterization. Note also that if κ and λ are, respectively, the dimensions of each $\mathbf{a}_j$ and $\mathbf{b}_i$, the total number of minimization parameters in (6) equals $m\kappa + n\lambda$ and is therefore large even for BA problems defined for rather short sequences. For example, in the case of projective reconstruction, $\kappa = 12$ and $\lambda = 4$ and, therefore, a moderately sized BA problem defined by, say, 1000 points projecting on each of 30 images involves 4360 variables. Sample dimensions of real Euclidean reconstruction problems that are employed in the experimental results section of this article can be found in the first columns of Table I.

BA can be cast as a nonlinear minimization problem as follows. A parameter vector $\mathbf{P} \in \mathcal{R}^M$ is defined by all parameters describing the m projection matrices and the n 3D points in Equation (6), namely, $\mathbf{P} = (\mathbf{a}_1^T, \ldots, \mathbf{a}_m^T, \ldots, \mathbf{b}_1^T, \ldots, \mathbf{b}_n^T)^T$. A measurement vector $\mathbf{X} \in \mathcal{R}^N$ is made up of the measured image point coordinates across all cameras:

$$\mathbf{X} = \left(\mathbf{x}_{11}^T, \ldots, \mathbf{x}_{1m}^T,\ \mathbf{x}_{21}^T, \ldots, \mathbf{x}_{2m}^T,\ \ldots,\ \mathbf{x}_{n1}^T, \ldots, \mathbf{x}_{nm}^T\right)^T. \tag{7}$$

Let $\mathbf{P}_0$ be an initial parameter estimate and $\Sigma_{\mathbf{X}}$ be the covariance matrix expressing the uncertainty of the measured vector $\mathbf{X}$; in the absence of any further knowledge, $\Sigma_{\mathbf{X}}$ is taken equal to the identity matrix. For each parameter vector, an estimated measurement vector $\hat{\mathbf{X}}$ is generated by a functional relation $\hat{\mathbf{X}} = f(\mathbf{P})$, defined by

$$\hat{\mathbf{X}} = \left(\hat{\mathbf{x}}_{11}^T, \ldots, \hat{\mathbf{x}}_{1m}^T,\ \hat{\mathbf{x}}_{21}^T, \ldots, \hat{\mathbf{x}}_{2m}^T,\ \ldots,\ \hat{\mathbf{x}}_{n1}^T, \ldots, \hat{\mathbf{x}}_{nm}^T\right)^T, \tag{8}$$

with $\hat{\mathbf{x}}_{ij} = \mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i)$.

Thus, BA corresponds to minimizing the squared $\Sigma_{\mathbf{X}}^{-1}$-norm (i.e., Mahalanobis distance) $\epsilon^T \Sigma_{\mathbf{X}}^{-1} \epsilon$, $\epsilon = \mathbf{X} - \hat{\mathbf{X}}$ over $\mathbf{P}$. Evidently, this minimization problem can be solved by employing the LM nonlinear least-squares algorithm, which calls for repeatedly solving the augmented weighted normal equations

$$(\mathbf{J}^T \Sigma_{\mathbf{X}}^{-1} \mathbf{J} + \mu \mathbf{I})\delta = \mathbf{J}^T \Sigma_{\mathbf{X}}^{-1} \epsilon, \tag{9}$$

where $\mathbf{J}$ is the Jacobian of f and δ is the sought update to the parameter vector $\mathbf{P}$. As will be demonstrated below, the normal equations in Equation (9) have a regular sparse block structure that results from the lack of interaction between parameters of different cameras and different 3D points. To keep the demonstration manageable, a case with small n and m is worked out in detail; however, as will later become apparent, the results are straightforward to generalize to arbitrary numbers of 3D points and cameras.

Assume that $n = 4$ points are visible in $m = 3$ views. The measurement vector is $\mathbf{X} = (\mathbf{x}_{11}^T,\ \mathbf{x}_{12}^T,\ \mathbf{x}_{13}^T,\ \mathbf{x}_{21}^T,\ \mathbf{x}_{22}^T,\ \mathbf{x}_{23}^T,\ \mathbf{x}_{31}^T,\ \mathbf{x}_{32}^T,\ \mathbf{x}_{33}^T,\ \mathbf{x}_{41}^T,\ \mathbf{x}_{42}^T,\ \mathbf{x}_{43}^T)^T$. The parameter vector is given by $\mathbf{P} = (\mathbf{a}_1^T,\ \mathbf{a}_2^T,\ \mathbf{a}_3^T,\ \mathbf{b}_1^T,\ \mathbf{b}_2^T,\ \mathbf{b}_3^T,\ \mathbf{b}_4^T)^T$. Notice that $\frac{\partial \hat{\mathbf{x}}_{ij}}{\partial \mathbf{a}_k} = \mathbf{0},\ \forall\ j \neq k$ and $\frac{\partial \hat{\mathbf{x}}_{ij}}{\partial \mathbf{b}_k} = \mathbf{0},\ \forall\ i \neq k$. Let $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ denote $\frac{\partial \hat{\mathbf{x}}_{ij}}{\partial \mathbf{a}_j}$ and $\frac{\partial \hat{\mathbf{x}}_{ij}}{\partial \mathbf{b}_i}$, respectively. The LM updating vector δ can be partitioned into camera and structure parameters as $(\delta_{\mathbf{a}}^T,\ \delta_{\mathbf{b}}^T)^T$ and further as $(\delta_{\mathbf{a}_1}^T, \delta_{\mathbf{a}_2}^T, \delta_{\mathbf{a}_3}^T, \delta_{\mathbf{b}_1}^T, \delta_{\mathbf{b}_2}^T, \delta_{\mathbf{b}_3}^T, \delta_{\mathbf{b}_4}^T)^T$. The remainder of this section is devoted to elaborating a scheme for efficiently solving the normal equations arising in LM minimization by taking advantage of their sparse structure.

Employing the notation for the derivatives introduced in the previous paragraph, we can write the Jacobian $\mathbf{J}$ as

$$\frac{\partial \mathbf{X}}{\partial \mathbf{P}} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{11} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{12} & \mathbf{0} & \mathbf{B}_{12} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{13} & \mathbf{B}_{13} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{21} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{22} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{23} & \mathbf{0} & \mathbf{B}_{23} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{31} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{31} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{32} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{33} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{33} & \mathbf{0} \\ \mathbf{A}_{41} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{41} \\ \mathbf{0} & \mathbf{A}_{42} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{42} \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{43} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{43} \end{pmatrix}. \tag{10}$$

Equation (10) clearly reveals the sparse nature of the matrix $\mathbf{J}$. It is due to $\mathbf{J}$'s sparseness that the normal equations are themselves sparse. Let the covariance matrix for the complete measurement vector be the block diagonal matrix $\Sigma_{\mathbf{X}} =$

$$diag(\Sigma_{\mathbf{x}_{11}},\ \Sigma_{\mathbf{x}_{12}},\ \Sigma_{\mathbf{x}_{13}},\ \Sigma_{\mathbf{x}_{21}},\ \Sigma_{\mathbf{x}_{22}},\ \Sigma_{\mathbf{x}_{23}},\ \Sigma_{\mathbf{x}_{31}},\ \Sigma_{\mathbf{x}_{32}},\ \Sigma_{\mathbf{x}_{33}},\ \Sigma_{\mathbf{x}_{41}},\ \Sigma_{\mathbf{x}_{42}},\ \Sigma_{\mathbf{x}_{43}}). \tag{11}$$

By substituting $\mathbf{J}$ and $\Sigma_{\mathbf{X}}^{-1}$ from Equations (10) and (11) and by denoting

$$\mathbf{U}_j \equiv \sum_{i=1}^{4} \mathbf{A}_{ij}^T \Sigma_{\mathbf{x}_{ij}}^{-1} \mathbf{A}_{ij},\ \mathbf{V}_i \equiv \sum_{j=1}^{3} \mathbf{B}_{ij}^T \Sigma_{\mathbf{x}_{ij}}^{-1} \mathbf{B}_{ij},\ \mathbf{W}_{ij} \equiv \mathbf{A}_{ij}^T \Sigma_{\mathbf{x}_{ij}}^{-1} \mathbf{B}_{ij}, \tag{12}$$

we can express the matrix product in the left-hand side of Equation (9) as

$$\begin{pmatrix} \mathbf{U}_1 & \mathbf{0} & \mathbf{0} & \mathbf{W}_{11} & \mathbf{W}_{21} & \mathbf{W}_{31} & \mathbf{W}_{41} \\ \mathbf{0} & \mathbf{U}_2 & \mathbf{0} & \mathbf{W}_{12} & \mathbf{W}_{22} & \mathbf{W}_{32} & \mathbf{W}_{42} \\ \mathbf{0} & \mathbf{0} & \mathbf{U}_3 & \mathbf{W}_{13} & \mathbf{W}_{23} & \mathbf{W}_{33} & \mathbf{W}_{43} \\ \mathbf{W}_{11}^T & \mathbf{W}_{12}^T & \mathbf{W}_{13}^T & \mathbf{V}_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{W}_{21}^T & \mathbf{W}_{22}^T & \mathbf{W}_{23}^T & \mathbf{0} & \mathbf{V}_2 & \mathbf{0} & \mathbf{0} \\ \mathbf{W}_{31}^T & \mathbf{W}_{32}^T & \mathbf{W}_{33}^T & \mathbf{0} & \mathbf{0} & \mathbf{V}_3 & \mathbf{0} \\ \mathbf{W}_{41}^T & \mathbf{W}_{42}^T & \mathbf{W}_{43}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{V}_4 \end{pmatrix}. \tag{13}$$

Also, using Equations (10) and (11), we can expand the right-hand side of Equation (9) as

$$\left(\sum_{i=1}^{4} \left(\mathbf{A}_{i1}^T \Sigma_{\mathbf{x}_{i1}}^{-1} \epsilon_{i1}\right)^T,\quad \sum_{i=1}^{4} \left(\mathbf{A}_{i2}^T \Sigma_{\mathbf{x}_{i2}}^{-1} \epsilon_{i2}\right)^T,\ \sum_{i=1}^{4} \left(\mathbf{A}_{i3}^T \Sigma_{\mathbf{x}_{i3}}^{-1} \epsilon_{i3}\right)^T, \right. \tag{14}$$

$$\left. \sum_{j=1}^{3} \left(\mathbf{B}_{1j}^T \Sigma_{\mathbf{x}_{1j}}^{-1} \epsilon_{1j}\right)^T,\quad \sum_{j=1}^{3} \left(\mathbf{B}_{2j}^T \Sigma_{\mathbf{x}_{2j}}^{-1} \epsilon_{2j}\right)^T,\ \sum_{j=1}^{3} \left(\mathbf{B}_{3j}^T \Sigma_{\mathbf{x}_{3j}}^{-1} \epsilon_{3j}\right)^T,\ \sum_{j=1}^{3} \left(\mathbf{B}_{4j}^T \Sigma_{\mathbf{x}_{4j}}^{-1} \epsilon_{4j}\right)^T \right)^T.$$

Letting

$$\epsilon_{\mathbf{a}_j} \equiv \sum_{i=1}^{4} \mathbf{A}_{ij}^T \Sigma_{\mathbf{x}_{ij}}^{-1} \epsilon_{ij},\ \epsilon_{\mathbf{b}_i} \equiv \sum_{j=1}^{3} \mathbf{B}_{ij}^T \Sigma_{\mathbf{x}_{ij}}^{-1} \epsilon_{ij},\ \text{with } \epsilon_{ij} \equiv \mathbf{x}_{ij} - \hat{\mathbf{x}}_{ij}\ \forall\, i, j, \tag{15}$$

we can abbreviate the vector in (14) to

$$\left(\epsilon_{\mathbf{a}_1}^T,\ \epsilon_{\mathbf{a}_2}^T,\ \epsilon_{\mathbf{a}_3}^T,\ \epsilon_{\mathbf{b}_1}^T,\ \epsilon_{\mathbf{b}_2}^T,\ \epsilon_{\mathbf{b}_3}^T,\ \epsilon_{\mathbf{b}_4}^T\right)^T. \tag{16}$$

By substituting the expressions for $\mathbf{J}^T \Sigma_{\mathbf{X}}^{-1} \mathbf{J}$ and $\mathbf{J}^T \Sigma_{\mathbf{X}}^{-1} \epsilon$ from (13) and (16), we find that the normal Equations (9) become

$$\left(\begin{array}{ccc|cccc} \mathbf{U}_1 & \mathbf{0} & \mathbf{0} & \mathbf{W}_{11} & \mathbf{W}_{21} & \mathbf{W}_{31} & \mathbf{W}_{41} \\ \mathbf{0} & \mathbf{U}_2 & \mathbf{0} & \mathbf{W}_{12} & \mathbf{W}_{22} & \mathbf{W}_{32} & \mathbf{W}_{42} \\ \mathbf{0} & \mathbf{0} & \mathbf{U}_3 & \mathbf{W}_{13} & \mathbf{W}_{23} & \mathbf{W}_{33} & \mathbf{W}_{43} \\ \hline \mathbf{W}_{11}^T & \mathbf{W}_{12}^T & \mathbf{W}_{13}^T & \mathbf{V}_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{W}_{21}^T & \mathbf{W}_{22}^T & \mathbf{W}_{23}^T & \mathbf{0} & \mathbf{V}_2 & \mathbf{0} & \mathbf{0} \\ \mathbf{W}_{31}^T & \mathbf{W}_{32}^T & \mathbf{W}_{33}^T & \mathbf{0} & \mathbf{0} & \mathbf{V}_3 & \mathbf{0} \\ \mathbf{W}_{41}^T & \mathbf{W}_{42}^T & \mathbf{W}_{43}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{V}_4 \end{array}\right) \begin{pmatrix} \delta_{\mathbf{a}_1} \\ \delta_{\mathbf{a}_2} \\ \delta_{\mathbf{a}_3} \\ \delta_{\mathbf{b}_1} \\ \delta_{\mathbf{b}_2} \\ \delta_{\mathbf{b}_3} \\ \delta_{\mathbf{b}_4} \end{pmatrix} = \begin{pmatrix} \epsilon_{\mathbf{a}_1} \\ \epsilon_{\mathbf{a}_2} \\ \epsilon_{\mathbf{a}_3} \\ \epsilon_{\mathbf{b}_1} \\ \epsilon_{\mathbf{b}_2} \\ \epsilon_{\mathbf{b}_3} \\ \epsilon_{\mathbf{b}_4} \end{pmatrix}. \tag{17}$$

Denoting the upper left, lower right, and upper right parts of the matrix in Equation (17), respectively, with $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$, allows us to further compact the augmented normal equations to

$$\begin{pmatrix} \mathbf{U}^* & \mathbf{W} \\ \mathbf{W}^T & \mathbf{V}^* \end{pmatrix} \begin{pmatrix} \delta_{\mathbf{a}} \\ \delta_{\mathbf{b}} \end{pmatrix} = \begin{pmatrix} \epsilon_{\mathbf{a}} \\ \epsilon_{\mathbf{b}} \end{pmatrix}, \tag{18}$$

where * designates the augmentation of the diagonal elements of $\mathbf{U}$ and $\mathbf{V}$. Left multiplication of Equation (18) by the block matrix

$$\begin{pmatrix} \mathbf{I} & -\mathbf{W}\,\mathbf{V}^{*-1} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \tag{19}$$

results in

$$\begin{pmatrix} \mathbf{U}^* - \mathbf{W}\,\mathbf{V}^{*-1}\,\mathbf{W}^T & \mathbf{0} \\ \mathbf{W}^T & \mathbf{V}^* \end{pmatrix} \begin{pmatrix} \delta_{\mathbf{a}} \\ \delta_{\mathbf{b}} \end{pmatrix} = \begin{pmatrix} \epsilon_{\mathbf{a}} - \mathbf{W}\,\mathbf{V}^{*-1}\,\epsilon_{\mathbf{b}} \\ \epsilon_{\mathbf{b}} \end{pmatrix}. \tag{20}$$

Since the top right block of the above left hand matrix is zero, the dependence of the camera parameters on the structure parameters has been eliminated in Equation (20) and, therefore, $\delta_{\mathbf{a}}$ can be determined from its top half, which is

$$(\mathbf{U}^* - \mathbf{W}\,\mathbf{V}^{*-1}\,\mathbf{W}^T)\,\delta_{\mathbf{a}} = \epsilon_{\mathbf{a}} - \mathbf{W}\,\mathbf{V}^{*-1}\,\epsilon_{\mathbf{b}}. \tag{21}$$

Matrix $\mathbf{S} \equiv \mathbf{U}^* - \mathbf{W}\,\mathbf{V}^{*-1}\,\mathbf{W}^T$ is the Schur complement of $\mathbf{V}^*$ in the left-hand side matrix of Equation (18). Since the Schur complement of a symmetric positive definite matrix is itself symmetric and positive definite [Prasolov 1994], the system of Equation (21) can be efficiently solved using the Cholesky factorization of $\mathbf{S}$. Having solved for $\delta_{\mathbf{a}}$, we can compute $\delta_{\mathbf{b}}$ by back substitution into the bottom half of Equation (20), which yields

$$\mathbf{V}^*\,\delta_{\mathbf{b}} = \epsilon_{\mathbf{b}} - \mathbf{W}^T\,\delta_{\mathbf{a}}. \tag{22}$$

The choice of solving first for $\delta_{\mathbf{a}}$ and then for $\delta_{\mathbf{b}}$ is justified by the fact that the total number of camera parameters is in general much smaller than the total number of structure parameters. Therefore, Equation (21) involves the solution of smaller systems that can be carried out with considerably fewer computations. Observing that $\mathbf{V}_i^{*-1}$ is equal to

$$\begin{pmatrix} \mathbf{V}_1^{*-1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_2^{*-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{V}_3^{*-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{V}_4^{*-1} \end{pmatrix} \tag{23}$$

and substituting $\mathbf{U}^*$, $\mathbf{W}$, and $\mathbf{V}^*$ with their corresponding blocks from Equation (17), matrix $\mathbf{S}$ in the left hand side of Equation (21) can be written as

$$\mathbf{S} = \begin{pmatrix} \mathbf{U}_1^* - \sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i1}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i2}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i3}^T \\ -\sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i1}^T & \mathbf{U}_2^* - \sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i2}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i3}^T \\ -\sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i1}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i2}^T & \mathbf{U}_3^* - \sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i3}^T \end{pmatrix}, \tag{24}$$

where $\mathbf{Y}_{ij} = \mathbf{W}_{ij}\mathbf{V}_i^{*-1}$. Further, the right-hand side of Equation (21) equals

$$\epsilon_{\mathbf{a}} - \left(\sum_{i=1}^{4} (\mathbf{Y}_{i1}\, \epsilon_{\mathbf{b}_i})^T,\ \sum_{i=1}^{4} (\mathbf{Y}_{i2}\, \epsilon_{\mathbf{b}_i})^T,\ \sum_{i=1}^{4} (\mathbf{Y}_{i3}\, \epsilon_{\mathbf{b}_i})^T \right)^T. \tag{25}$$

By combining (24) and (25), we can compute $\delta_{\mathbf{a}}$ by solving the system

$$\begin{pmatrix} \mathbf{U}_1^* - \sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i1}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i2}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i3}^T \\ -\sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i1}^T & \mathbf{U}_2^* - \sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i2}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i3}^T \\ -\sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i1}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i2}^T & \mathbf{U}_3^* - \sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i3}^T \end{pmatrix} \begin{pmatrix} \delta_{\mathbf{a}_1} \\ \delta_{\mathbf{a}_2} \\ \delta_{\mathbf{a}_3} \end{pmatrix} = \begin{pmatrix} \epsilon_{\mathbf{a}_1} - \sum_{i=1}^{4} \mathbf{Y}_{i1}\, \epsilon_{\mathbf{b}_i} \\ \epsilon_{\mathbf{a}_2} - \sum_{i=1}^{4} \mathbf{Y}_{i2}\, \epsilon_{\mathbf{b}_i} \\ \epsilon_{\mathbf{a}_3} - \sum_{i=1}^{4} \mathbf{Y}_{i3}\, \epsilon_{\mathbf{b}_i} \end{pmatrix}. \tag{26}$$

Finally, left multiplication of Equation (22) by $\mathbf{V}^{*-1}$ from (23) yields $\delta_{\mathbf{b}}$ from $\delta_{\mathbf{a}}$ as

$$\delta_{\mathbf{b}} = \begin{pmatrix} \mathbf{V}_1^{*-1} \left(\epsilon_{\mathbf{b}_1} - \sum_{j=1}^{3} \mathbf{W}_{1j}^T\, \delta_{\mathbf{a}_j} \right) \\ \mathbf{V}_2^{*-1} \left(\epsilon_{\mathbf{b}_2} - \sum_{j=1}^{3} \mathbf{W}_{2j}^T\, \delta_{\mathbf{a}_j} \right) \\ \mathbf{V}_3^{*-1} \left(\epsilon_{\mathbf{b}_3} - \sum_{j=1}^{3} \mathbf{W}_{3j}^T\, \delta_{\mathbf{a}_j} \right) \\ \mathbf{V}_4^{*-1} \left(\epsilon_{\mathbf{b}_4} - \sum_{j=1}^{3} \mathbf{W}_{4j}^T\, \delta_{\mathbf{a}_j} \right) \end{pmatrix}. \tag{27}$$

Input: The current parameter vector partitioned into m camera parameter vectors $\mathbf{a}_j$ and n 3D point parameter vectors $\mathbf{b}_i$, a function $\mathbf{Q}$ employing the $\mathbf{a}_j$ and $\mathbf{b}_i$ to compute the predicted projections $\hat{\mathbf{x}}_{ij}$ of the i-th point on the jth image, the observed image point locations $\mathbf{x}_{ij}$ and a damping term μ for LM.
Output: The solution δ to the normal equations involved in LM-based bundle adjustment.
Algorithm:

Compute the derivative matrices $\mathbf{A}_{ij} := \frac{\partial \hat{\mathbf{x}}_{ij}}{\partial \mathbf{a}_j} = \frac{\partial \mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i)}{\partial \mathbf{a}_j}$,
$\mathbf{B}_{ij} := \frac{\partial \hat{\mathbf{x}}_{ij}}{\partial \mathbf{b}_i} = \frac{\partial \mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i)}{\partial \mathbf{b}_i}$ and the error vectors $\epsilon_{ij} := \mathbf{x}_{ij} - \hat{\mathbf{x}}_{ij}$,
where i and j assume values from $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$, respectively.

Compute the following auxiliary variables:

$$\mathbf{U}_j := \sum_i \mathbf{A}_{ij}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{A}_{ij} \qquad \mathbf{V}_i := \sum_j \mathbf{B}_{ij}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{B}_{ij} \qquad \mathbf{W}_{ij} := \mathbf{A}_{ij}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{B}_{ij},$$
$$\epsilon_{\mathbf{a}_j} := \sum_i \mathbf{A}_{ij}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \epsilon_{ij}, \qquad \epsilon_{\mathbf{b}_i} := \sum_j \mathbf{B}_{ij}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \epsilon_{ij}.$$

Augment $\mathbf{U}_j$ and $\mathbf{V}_i$ by adding μ to their diagonals to yield $\mathbf{U}_j^*$ and $\mathbf{V}_i^*$.

Compute $\mathbf{Y}_{ij} := \mathbf{W}_{ij} \mathbf{V}_i^{*\,-1}$.

Compute $\delta_{\mathbf{a}}$ from $\mathbf{S}\ (\delta_{\mathbf{a}_1}^T, \delta_{\mathbf{a}_2}^T, \ldots, \delta_{\mathbf{a}_m}^T)^T = (\mathbf{e}_1^T, \mathbf{e}_2^T, \ldots, \mathbf{e}_m^T)^T$,
where $\mathbf{S}$ is a matrix consisting of $m \times m$ blocks; block jk is defined by
$\mathbf{S}_{jk} = \delta_{jk} \mathbf{U}_j^* - \sum_i \mathbf{Y}_{ij} \mathbf{W}_{ik}^T$, where δ_{jk} is Kronecker's delta
and
$\mathbf{e}_j = \epsilon_{\mathbf{a}_j} - \sum_i \mathbf{Y}_{ij}\ \epsilon_{\mathbf{b}_i}$.

Compute each $\delta_{\mathbf{b}_i}$ from the equation $\delta_{\mathbf{b}_i} = \mathbf{V}_i^{*\,-1}\ (\epsilon_{\mathbf{b}_i} - \sum_j \mathbf{W}_{ij}^T\ \delta_{\mathbf{a}_j})$.

Form δ as $(\delta_{\mathbf{a}}^T,\ \delta_{\mathbf{b}}^T)^T$.

Fig. 4. Algorithm for solving the sparse, augmented normal equations arising in generic bundle adjustment. Combined with the algorithm portrayed in Figure 2, yields a sparse BA algorithm.

At this point, it should be evident that the approach for solving the normal equations that was illustrated above can be directly generalized to arbitrary n and m. Note also that if a point k does not appear in an image l then $\mathbf{A}_{kl} = \mathbf{0}$ and $\mathbf{B}_{kl} = \mathbf{0}$. Hence, index i in the summations appearing in the definitions of $\mathbf{U}_j$ and $\epsilon_{\mathbf{a}_j}$ (see Equations (12) and (15)) runs through all points appearing in the specified camera j. Similarly, index j in the definitions of $\mathbf{V}_i$ and $\epsilon_{\mathbf{b}_i}$ runs through all cameras to which the given point i is projected. Figure 4 summarizes the general procedure for solving the sparse normal equations involved in the LM algorithm. This procedure can be embedded in the LM algorithm of Section 3 at the point indicated by the rectangular box in Figure 2, leading to a sparse bundle adjustment algorithm.

5. IMPLEMENTATION DETAILS

This section provides some details regarding the practical implementation of the sparse bundle adjustment algorithm sketched in Sections 3 and 4. The primary emphases of the design were on flexibility and performance efficiency.

To cater for different user needs, expert and simple drivers to sparse bundle adjustment have been developed. The expert drivers, discussed in Section 5.1, are aimed at the highest performance but require that the user understands and conforms to certain rules regarding the internal representation of the data objects involved in sparse bundle adjustment. On the other hand, the simple drivers presented in Section 5.2 are designed for the less knowledgeable user who is willing to trade some potential loss in performance for increased ease of use. The reader should be cautioned that reported details refer to `sba` v.1.5 (rel. July 08); future versions may differ slightly.

5.1 Expert Drivers

Let us begin by considering the matrices $\mathbf{A}$ and $\mathbf{B}$ consisting, respectivel, of the blocks $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ defined in the algorithm of Figure 4. Note that, if a point i does not appear in image j, then $\mathbf{A}_{ij} = \mathbf{B}_{ij} = \mathbf{0}$, which implies that both $\mathbf{A}$ and $\mathbf{B}$ are sparse. For instance, referring to the sample problem with $n = 4$ and $m = 3$ outlined in Section 4, assuming that point 1 is not visible in image 3 and points 2, 3, 4 are not visible in image 1 implies that $\mathbf{A}_{13} = \mathbf{A}_{21} = \mathbf{A}_{31} = \mathbf{A}_{41} = \mathbf{0}$. Hence, the corresponding $\mathbf{A}$ in block form is as shown in the left part of (28); $\mathbf{B}$ has a similar structure.

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} \\ \mathbf{0} & \mathbf{A}_{42} & \mathbf{A}_{43} \end{pmatrix}, \quad \mathcal{I} = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 2 & 3 \\ -1 & 4 & 5 \\ -1 & 6 & 7 \end{pmatrix}. \tag{28}$$

While storing all zero elements of $\mathbf{A}$ and $\mathbf{B}$ is acceptable for moderately sized BA problems, it becomes inefficient in terms of memory use when the numbers of 3D points and cameras are large. To save memory, $\mathbf{A}$ is stored in a contiguous memory buffer by arranging its non-zero blocks $\mathbf{A}_{ij}$ in succession and setting up a matrix $\mathcal{I}$ whose (i, j) element contains the index (i.e., offset) k of the block in the memory buffer that has been allocated to $\mathbf{A}_{ij}$. Matrix $\mathcal{I}$ for the sample $\mathbf{A}$ is shown in the right part of (28). Indices stored in $\mathcal{I}$ conform to the C convention and start from zero. Elements equal to -1 indicate that the corresponding $\mathbf{A}_{ij}$ is zero and therefore does not need to be stored. Clearly, $\mathcal{I}$ is itself a sparse matrix and therefore special techniques are used for storing it.

To reduce the memory requirements of large sparse matrices, researchers in numerical linear algebra have devised various memory storage schemes. Those schemes allocate a contiguous memory segment for storing the nonzero matrix elements along with some additional information for keeping track where the stored elements fit into the full matrix. In the context of this work, we have chosen to represent sparse matrices using the compressed row storage (CRS) format [Barrett et al. 1994], which is described next. CRS makes no assumptions regarding the sparsity structure of the matrix and does not store any unnecessary elements. It employs contiguous memory locations to store the following vectors: the `val` vector which stores the values of the nonzero matrix elements in row-major order, the `colidx` vector that stores the column indices of the elements in the `val` vector, and the `rowptr` vector which stores the locations in the `val` vector, that start a row. In other words, if `val[k]=a[i][j]` then `colidx[k]=j` and `rowptr[i] <= k < rowptr[i+1]`. To simplify algorithms operating on CRS structures, `rowptr` by convention contains an extra element at its end, equal to the number of nonzero array elements. As an example, the CRS vectors for the 4×3 matrix $\mathcal{I}$, with 8 nonzero elements defined above, are shown below. Again, array indices conform to the C convention and start from zero:

$$\begin{aligned} \texttt{val}: & \ (0, 1, 2, 3, 4, 5, 6, 7), \\ \texttt{colidx}: & \ (0, 1, 1, 2, 1, 2, 1, 2), \\ \texttt{rowptr}: & \ (0, 2, 4, 6, 8). \end{aligned}$$

Thus, the `sba` routines store $\mathbf{A}$ by keeping its nonzero $\mathbf{A}_{ij}$ blocks in a contiguous memory buffer and using a CRS structure to store $\mathcal{I}$. $\mathbf{B}$ is stored in a similar manner, and, since $\mathbf{B}_{ij}$ is zero whenever $\mathbf{A}_{ij}$ is zero, $\mathcal{I}$ can be reused to provide the mapping between block pair indices (i, j) for each $\mathbf{B}_{ij}$ and the corresponding contiguous memory block indices. Note also that blocks $\mathbf{W}_{ij}$ and $\mathbf{Y}_{ij}$ defined in the algorithm of Figure 4 are zero if either of $\mathbf{A}_{ij}$ or $\mathbf{B}_{ij}$ is zero. Therefore, the matrices $\mathbf{W}$ and $\mathbf{Y}$ consisting of blocks $\mathbf{W}_{ij}$ and $\mathbf{Y}_{ij}$ are also sparse and can be stored in memory as explained for $\mathbf{A}$ and $\mathbf{B}$, again using the same $\mathcal{I}$ to hold the indices mapping. The aforementioned strategy for storing matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{W}$, and $\mathbf{Y}$ makes efficient use of the processor's cache since the elements of nonzero blocks are kept in consecutive memory locations, which are very likely to fit in a single cache block while they are being processed. Also, note that if point i does not appear in image j, then elements $\mathbf{x}_{ij}$ and $\hat{\mathbf{x}}_{ij}$ are missing from $\mathbf{X}$ and $\hat{\mathbf{X}}$ in Equation (7) and (8), respectively. The CRS structure holding $\mathcal{I}$ is once again employed to provide the mapping between $\mathbf{x}_{ij}$, $\hat{\mathbf{x}}_{ij}$ and their actual storage locations in $\mathbf{X}$, $\hat{\mathbf{X}}$. The CRS format is represented by `sba` in C/C++ using a structure with the following declaration:

```
struct sba_crsm{
   int nr, nc;   /* #rows, #columns for the sparse matrix */
   int nnz;      /* number of non-zero array elements */
   int *val;     /* storage for non-zero array elements. size: nnz */
   int *colidx;  /* column indices of non-zero elements. size: nnz */
   int *rowptr;  /* nr+1 locations in val that start a row. rowptr[nr]=nnz. */
};
```

Nonetheless, when solving large BA problems, the amount of memory that is necessary for storing the matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{W}$, and $\mathbf{Y}$ defined in the algorithm of Figure 4 might be excessive, even if sparse representations are employed for them. Close examination of the algorithm reveals that matrices $\mathbf{A}$ and $\mathbf{B}$ are not needed anymore after computing the auxiliary variables. Therefore, `sba` saves memory by making the nonzero blocks of derivative matrices $\mathbf{A}$ and $\mathbf{B}$ for the Jacobian share the same storage with the nonzero blocks of matrix $\mathbf{W}$. At each iteration, matrices $\mathbf{A}$ and $\mathbf{B}$ are computed in a working memory that is later overwritten during the computation of $\mathbf{W}$. This is the reason for requiring that blocks $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ are ordered back-to-back by the function evaluating the Jacobian (see also the description of argument `fjac` below). Besides facilitating the reduction of the required amount of memory, this choice improves locality during the computation of blocks $\mathbf{W}_{ij}$, thus reducing execution time. Memory requirements are further lowered by choosing not to store the full matrix consisting of the $\mathbf{Y}_{ij}$ blocks. Instead, for each image j the block column consisting of the nonzero $\mathbf{Y}_{ij}$, $i = 1, \ldots, n$ is stored and then used in the calculations involving $\mathbf{S}_{jk}$ and $\mathbf{e}_j$ (cf. Figure 4). This memory storage is reused when computing the $\mathbf{Y}_{ij}$ corresponding to other block columns. Yet another performance improvement concerns the computation of auxiliary matrices $\mathbf{U}_j$, $\mathbf{V}_i$, and $\mathbf{S}$, which is sped up by exploiting their symmetry and explicitly computing only their triangular parts.

An important observation concerning the matrix $\mathbf{S}$ from Figure 4 is that typical point tracks do not span the whole image sequence but rather extend over a relatively small number of successive images. This means that, especially in long image sequences, it is rare to have points that are tracked across all images, which in turn implies that it is very likely that certain images j and k have no visible points in common and thus the corresponding off-diagonal block jk of $\mathbf{S}$ is zero. In other words, matrix $\mathbf{S}$ can be sparse. For all but one of the sequences employed in the experiments reported in Section 6, the density of $\mathbf{S}$ (defined as the ratio of nonzero elements over the product of rows and columns) was at least 84%; actually five out of the total eight sequences used had a density of 100%. Therefore, considering that the memory savings of a sparse representation would be outweighted by the extra overhead for accessing the nonzero matrix elements, it was chosen to represent $\mathbf{S}$ as a dense matrix containing zeros. For very long sequences, however, $\mathbf{S}$ is expected to be much sparser and should be dealt with accordingly.

Covariances $\Sigma_{\mathbf{x}_{ij}}$ are accommodated by computing the upper triangular matrices $\mathbf{C}_{\mathbf{x}_{ij}}$ from the Cholesky decompositions of their inverses as $\Sigma_{\mathbf{x}_{ij}}^{-1} = \mathbf{C}_{\mathbf{x}_{ij}}^T \mathbf{C}_{\mathbf{x}_{ij}}$ and using the $\mathbf{C}_{\mathbf{x}_{ij}}$ at every iteration to weigh $\mathbf{A}_{ij}$, $\mathbf{B}_{ij}$ and ϵ_{ij} by left multiplication. Auxiliary variables such as $\mathbf{U}_j$ are actually computed as $\sum_i \mathbf{A}_{ij}^T \mathbf{A}_{ij}$; if the $\mathbf{A}_{ij}$ have been pre-weighted with the covariances, the computation for $\mathbf{U}_j$ is equivalent to $\sum_i (\mathbf{C}_{\mathbf{x}_{ij}} \mathbf{A}_{ij})^T \mathbf{C}_{\mathbf{x}_{ij}} \mathbf{A}_{ij} = \sum_i \mathbf{A}_{ij}^T \Sigma_{\mathbf{x}_{ij}}^{-1} \mathbf{A}_{ij}$. Variables $\mathbf{V}_i$, $\mathbf{W}_{ij}$, $\epsilon_{\mathbf{a}_j}$, and $\epsilon_{\mathbf{b}_i}$ are computed in a similar fashion. This strategy permits the incorporation of covariances with minimal overhead when they are all equal to identity matrices and in the opposite case, with fewer matrix multiplications than those that would be required by the straightforward computation suggested by the definitions of $\mathbf{U}_j$, $\mathbf{V}_i$, $\mathbf{W}_{ij}$, $\epsilon_{\mathbf{a}_j}$, and $\epsilon_{\mathbf{b}_i}$ in Figure 4.

Sparse BA is implemented in `sba` by the expert function `sba_motstr_levmar_x()`. The prototype declaration of `sba_motstr_levmar_x()` is supplied in the `sba.h` header file that defines `sba`'s API and reads as follows:

```
int sba_motstr_levmar_x(int n, int m, int mcon, char *vmask, double *p,
   int cnp, int pnp, double *x, double *covx, int mnp,
   void (*func)(double *p, struct sba_crsm *idxij, int *wk1, int *wk2, double *hx, void
     *adata),
   void (*fjac)(double *p, struct sba_crsm *idxij, int *wk1, int *wk2, double *jac, void
     *adata),
   void *adata, int itmax, int verbose, double opts[SBA_OPTSSZ], double
     info[SBA_INFOSZ]);
```

In case of successful termination, the function returns the number of iterations required for the minimization (≥ 0), and otherwise the negative constant SBA_ERROR. The linear system of Equation (21) is solved with the aid of Cholesky factorization, implemented using LAPACK routines DPOTRF and DPOTRS [Anderson et al. 1999]. For experimenting with other approaches (not necessarily relying on positive definiteness), sba also includes implementations of LAPACK-based linear system solvers employing LDLT, LU, QR, and SVD decompositions [Golub and van Loan 1996; Demmel 1997]. LAPACK can be substituted by any equivalent vendor library (e.g., ESSL, MKL, NAG, etc) that conforms to the API described in the LAPACK User's Guide. sba_motstr_levmar_x() implements a forward communication mechanism; its arguments are explained one by one in the following, where I and O denote input and output arguments, respectively:

—n: The number of 3D points. (I)

—m: The number of cameras (i.e., images). (I)

—mcon: The number of cameras (starting from the first) whose parameters should not be modified. All $\mathbf{A}_{ij}$ with $j <$ mcon are assumed to be zero. This is, for example, useful when the world's coordinate frame is aligned with that of the first camera, therefore the (projective) first camera matrix should be kept fixed to $\left[\mathbf{I} \mid \mathbf{0}\right]$. (I)

—vmask: Point visibility mask: vmask[(i-1)*m+j-1] equals 1 if point i is visible in image j, 0 otherwise. Note that in the preceding presentation points and images are numbered as $1, 2, \ldots$, whereas C array indices start from zero, hence the -1s in the expression (i-1)*m+j-1. The size of vmask is n*m. (I)

—p: On input, the initial parameter vector $\mathbf{P}_0 = (\mathbf{a}_1^T, \ldots, \mathbf{a}_m^T, \ldots, \mathbf{b}_1^T, \ldots, \mathbf{b}_n^T)^T$, where $\mathbf{a}_j$ are the parameters of image j and $\mathbf{b}_i$ are the parameters of point i. On output, the estimated minimizer. Its size is m*cnp + n*pnp. (I/O)

—cnp: The number of parameters defining a single camera. For example, a Euclidean camera parameterized using an angle-axis representation for rotation depends on six parameters (3 rotational + 3 translational). If quaternions are used for the rotations, the number of parameters increases to seven (i.e., 4 + 3). An affine camera can be modeled using seven parameters, while a fully projective one can be parameterized with eleven or, including the scale factor, twelve parameters. (I)

—pnp: The number of parameters defining a single 3D point: for example, three for Euclidean points, four for projective, etc. (I)

—x: The measurement vector $\mathbf{X}$ consisting of all image projections in the order $(\mathbf{x}_{11}^T, \ldots, \mathbf{x}_{1m}^T, \ldots, \mathbf{x}_{n1}^T, \ldots, \mathbf{x}_{nm}^T)^T$. For every point i that is not visible in image j (see vmask above), the corresponding $\mathbf{x}_{ij}$ is missing from x. Its size is NZ*mnp, where NZ $\leq$ n*m denotes the number of nonzeros in vmask and amounts to the total number of point projections in all images. (I)

—covx: The nonzero diagonal blocks of the covariance matrix $\Sigma_{\mathbf{X}}$ corresponding to the measurement vector $\mathbf{X}$. covx consists of the covariance matrices of image projections arranged as $\Sigma_{\mathbf{x}_{11}}, \ldots, \Sigma_{\mathbf{x}_{1m}}, \ldots, \Sigma_{\mathbf{x}_{n1}}, \ldots, \Sigma_{\mathbf{x}_{nm}}$, with each of the mnp*mnp matrices $\Sigma_{\mathbf{x}_{ij}}$ being stored in row-major order. Similarly to x above, for every point i that is not visible in image j, the corresponding $\Sigma_{\mathbf{x}_{ij}}$ is missing from covx. Its size is NZ*mnp*mnp, with NZ again denoting the number of nonzeros in vmask. A NULL value for covx forces identity matrices to be used for all $\Sigma_{\mathbf{x}_{ij}}$, omitting covariance information. (I)

—mnp: The number of parameters defining an image point (typically two). (I)

—func: The function computing the estimated measurement vector. Given an estimate of the parameters vector $\mathbf{P}$ in p, computes $\hat{\mathbf{X}}$ in hx by evaluating the parameterizing function $\mathbf{Q}()$ of (6) for all points and cameras. The measurement vector should be returned as $(\hat{\mathbf{x}}_{11}^T, \ldots, \hat{\mathbf{x}}_{1m}^T, \ldots, \hat{\mathbf{x}}_{n1}^T, \ldots, \hat{\mathbf{x}}_{nm}^T)^T$. Argument idxij is built up by sba_motstr_levmar_x() according to the information contained in its vmask argument. It specifies which points are visible in each image and provides the mapping between every $\hat{\mathbf{x}}_{ij}$ and its mnp-sized storage location in hx. Arguments wk1 and wk2 are arrays of size max(n, m) that have been allocated by the caller and can be used as working memory for the routines manipulating the idxij structure. Argument adata is identical to the so named argument of sba_motstr_levmar_x(), pointing to possibly additional data (see below). (I)

—fjac: The function evaluating in jac the sparse Jacobian $\mathbf{J}$ at p. $\mathbf{J}$ is made up of the derivatives of the parameterizing function $\mathbf{Q}()$ and should be laid out as $(\mathbf{A}_{11}, \mathbf{B}_{11}, \ldots, \mathbf{A}_{1m}, \mathbf{B}_{1m}, \ldots, \mathbf{A}_{n1}, \mathbf{B}_{n1}, \ldots, \mathbf{A}_{nm}, \mathbf{B}_{nm}$, where $\mathbf{A}_{ij} = \frac{\partial \mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i)}{\partial \mathbf{a}_j}$ and $\mathbf{B}_{ij} = \frac{\partial \mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i)}{\partial \mathbf{b}_i}$. Each of the $\mathbf{A}_{ij}$ (respectively $\mathbf{B}_{ij}$) blocks is made up of mnp*cnp (respectively mnp*pnp) elements and is stored in a row-major order, occupying a distinct storage block in jac. Note that the $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ for a certain image projection $\mathbf{x}_{ij}$ consist of mnp*(cnp+pnp) elements and are placed back-to-back in the jac array, which is comprised of exactly idxij->nnz pairs of $\mathbf{A}_{ij}$, $\mathbf{B}_{ij}$ blocks. The offset of each $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ in jac is determined through the idxij argument. The contents of idxij also specify which of the $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ are missing. Arguments wk1 and wk2 again point to working memory allocated by the caller. sba offers the option of automatically verifying the correctness of user-supplied Jacobians; please refer to the comments regarding itmax below. If the user specifies a NULL value for fjac, the Jacobian is approximated using forward finite differences on data provided by successive invocations of func. In order to reduce the total number of func invocations, the Jacobian is approximated using a scheme that computes several of its columns with a single func evaluation, exploiting its sparse structure as explained in Nocedal and Wright [1999], chapter 7. This scheme requires only cnp+pnp+1 func evaluations, that is, many fewer compared to the m*cnp+n*pnp+1 that would be required by the naive approach of computing a single column of the Jacobian per func evaluation. Still, the overhead associated with this finite difference Jacobian approximation might be significant. Therefore, such an approximation is primarily intended for use during initial testing and debugging and should be avoided when execution speed is a major concern, in which case the Jacobian should be computed analytically by a user-supplied function. (I)

—adata: Pointer to possibly additional data, passed unchanged to func and fjac. It is intended to facilitate accessing problem-specific data, avoiding direct use of global variables in the routines func and fjac. For example, a structure containing pointers to appropriate data structures can be set up and a pointer to it can be passed as the value of `adata` to `sba_motstr_levmar_x()`, which then passes it unchanged to each call of the user-supplied routines. This argument should be set to `NULL` if not needed. (I)

—`itmax`: Maximum number of Levenberg-Marquardt iterations (k_{max} in the algorithm of Figure 2). A zero value for `itmax` triggers verification of the user-supplied Jacobian followed by an immediate return with a value of zero. The correctness of the Jacobian is verified using an approach similar to that of MINPACK's CHKDER routine [Moré et al. 1980]. The verification routine prints suspicious gradients (i.e., Jacobian rows) to `stderr`. (I)

—`verbose`: Verbosity level. A value of zero specifies silent operation, larger values correspond to increasing verbosity levels. (I)

—`opts`: An array with `SBA_OPTSSZ` elements that specify the minimization options τ, ε_1, ε_2, ε_3, ε_4 for the Levenberg-Marquardt algorithm (see Figure 2). These are, respectively, the scale factor for the initial damping term and the stopping tolerance thresholds. `SBA_OPTSSZ` is a constant defined in `sba.h`. (I)

—`info`: Array of `SBA_INFOSZ` elements containing information regarding the outcome of the minimization. It can be set to `NULL` if not needed. `SBA_INFOSZ` is a constant defined in `sba.h`. (O)

—`info[0]`: $||\epsilon_{\mathbf{p}_0}||^2$, that is, the error at the initial parameters estimate. Note that `info[0]` divided by the total number of image point measurements (i.e., the number of nonzeros in `vmask`) corresponds to the initial mean squared reprojection error.

—`info[1-4]`: $(||\epsilon_{\mathbf{p}}||^2, ||\mathbf{J}^T\epsilon_{\mathbf{p}}||_\infty, ||\delta_{\mathbf{p}}||^2, \mu/\max_k\{[\mathbf{J}^T\mathbf{J}]_{kk}\})$, all computed at the final $\mathbf{p}$. Analogously to `info[0]`, `info[1]` divided by the number of image point measurements yields the final mean squared reprojection error.

—`info[5]`: Total number of iterations.

—`info[6]`: Reason for terminating:

1 : Stopped by small $||\mathbf{J}(\mathbf{p})^T\epsilon_{\mathbf{p}}||_\infty$.
2 : Stopped by small $||\delta_{\mathbf{p}}||$.
3 : Stopped by `itmax`.
4 : Stopped by small relative reduction in $||\epsilon_{\mathbf{p}}||$.
5 : Stopped by small $||\epsilon_{\mathbf{p}}||$.
6 : Stopped due to excessive failed attempts to increase damping for getting a positive definite normal equations matrix. Typically, this indicates a programming error in the user-supplied Jacobian.
7 : Stopped due to infinite values in the coordinates of the set of predicted projections $\hat{\mathbf{x}}_{ij}$. This signals a programming error in the user-supplied projection function `func`.

—`info[7]`: Total number of `func` evaluations.

—`info[8]`: Total number of `fjac` evaluations.

—`info[9]`: Total number of times that the augmented normal equations were solved. This is always larger that the number of iterations, since during a single LM iteration, several damping factors might be tried, each requiring the solution of the corresponding augmented normal equations (cf. the innermost loop of the algorithm in Figure 2).

The discussion of the arguments of `sba_motstr_levmar_x()` has made clear that all information pertaining to BA is selectable by its user: Any number of cameras and 3D points may be specified, each described by as many parameters as the user sees fit. The exact parameterization of motion and structure is defined by supplying appropriate `func` and `fjac` routines. Therefore, the user has complete freedom on the evaluation of the estimated measurement vector and its Jacobian. The user also has the ability to specify the visibility of point projections on an image basis.

Additionally, `sba` offers the expert function `sba_mot_levmar_x()` that minimizes the reprojection error with respect to the camera viewing parameters only. In other words, all 3D structure parameters are kept constant (therefore all $\mathbf{B}_{ij}$=$\mathbf{0}$) and only the camera motion/calibration parameters are modified. Strictly speaking, this function does not perform BA. Nevertheless, it is very useful when dealing with camera resectioning, that is, the problem of estimating the camera matrix from the 2D image projections of some 3D points that are assumed fixed and precisely known [Lu et al. 2000; Hartley 1993]. The prototype of `sba_mot_levmar_x()` is the following:

```
int sba_mot_levmar_x(int n, int m, int mcon, char *vmask, double *p,
    int cnp, double *x, double *covx, int mnp,
    void (*func)(double *p, struct sba_crsm *idxij, int *wk1, int *wk2, double *hx, void
     *adata),
    void (*fjac)(double *p, struct sba_crsm *idxij, int *wk1, int *wk2, double *jac, void
      *adata),
    void *adata, int itmax, int verbose, double opts[SBA_OPTSSZ],
      double info[SBA_INFOSZ]);
```

Function `sba_mot_levmar_x()` implements the algorithm resulting from that in Figure 4 after setting $\mathbf{B}_{ij} = \mathbf{V}_i = \mathbf{W}_{ij} = \mathbf{Y}_{ij} = \mathbf{0}$. Notice that in this case, the augmented normal equations of Equation (18) are simplified to a set of linear systems $\mathbf{U}_j^* \, \delta_{\mathbf{a}_j} = \epsilon_{\mathbf{a}_j}$, which, due to the positive definiteness of all $\mathbf{U}_j^*$, can be solved with the aid of Cholesky factorizations. Since the arguments of `sba_mot_levmar_x()` have the same meaning as their counterparts in `sba_motstr_levmar_x()`, no further explanation is given here.

Finally, `sba` includes the expert function `sba_str_levmar_x()`, which is in a sense complementary to `sba_mot_levmar_x()`. More specifically, `sba_str_levmar_x()` keeps the camera viewing parameters unchanged and minimizes the reprojection error with respect to the scene structure parameters only. This function is, for example, useful when reconstructing 3D points seen in a set of extrinsically calibrated images. It implements the algorithm resulting from that in Figure 4 after setting $\mathbf{A}_{ij} = \mathbf{U}_j = \mathbf{W}_{ij} = \mathbf{Y}_{ij} = \mathbf{0}$, hence reducing the normal equations to the set $\mathbf{V}_i^* \, \delta_{\mathbf{b}_i} = \epsilon_{\mathbf{b}_i}$. The prototype of function `sba_str_levmar_x()` is as follows:

```
int sba_str_levmar_x(int n, int m, char *vmask, double *p,
   int pnp, double *x, double *covx, int mnp,
   void (*func)(double *p, struct sba_crsm *idxij, int *wk1, int *wk2, double *hx, void
    *adata),
void (*fjac)(double *p, struct sba_crsm *idxij, int *wk1, int *wk2, double *jac, void
 *adata),
void *adata, int itmax, int verbose, double opts[SBA_OPTSSZ], double
 info[SBA_INFOSZ]);
```

Again, the arguments of `sba_str_levmar_x()` have the same meaning as their counterparts in `sba_motstr_levmar_x()`.

5.2 Simple Drivers

For users who are unwilling to spend much time understanding its inner workings, `sba` offers three simple drivers, namely, `sba_motstr_levmar()`, `sba_mot_levmar()`, and `sba_str_levmar()`, which are implemented as wrappers around the expert drivers `sba_motstr_levmar_x()`, `sba_mot_levmar_x()`, and `sba_str_levmar_x()`, respectively. They differ from the latter in that, instead of accepting arguments for estimating the whole measurement vector and its sparse Jacobian (i.e., `func` and `fjac`), they should be provided with routines to estimate a single image projection and its Jacobian (i.e., `proj` and `projac`). The wrappers then estimate the measurement vector and its sparse Jacobian by repeatedly invoking `proj` and `projac` for all points and cameras. Thus, at the moderate cost induced by the extra function calls, the simple drivers free the user from worrying about how the measurement vector and its sparse Jacobian are laid out in memory internally.

The prototype of `sba_motstr_levmar()` is

```
int sba_motstr_levmar(int n, int m, int mcon, char *vmask, double *p,
  int cnp, int pnp, double *x, double *covx, int mnp,
  void (*proj)(int j, int i, double *aj, double *bi, double *xij, void *adata),
  void (*projac)(int j, int i, double *aj, double *bi, double *Aij, double *Bij, void *adata),
  void *adata, int itmax, int verbose, double opts[SBA_OPTSSZ], double
    info[SBA_INFOSZ]);
```

that of `sba_mot_levmar()`:

```
int sba_mot_levmar(int n, int m, int mcon, char *vmask, double *p,
  int cnp, double *x, double *covx, int mnp,
  void (*proj)(int j, int i, double *aj, double *xij, void *adata),
  void (*projac)(int j, int i, double *aj, double *Aij, void *adata),
  void *adata, int itmax, int verbose, double opts[SBA_OPTSSZ], double
    info[SBA_INFOSZ]);
```

and that of `sba_str_levmar()`:

```
int sba_str_levmar(int n, int m, char *vmask, double *p,
  int pnp, double *x, double *covx, int mnp,
  void (*proj)(int j, int i, double *bi, double *xij, void *adata),
  void (*projac)(int j, int i, double *bi, double *Bij, void *adata),
  void *adata, int itmax, int verbose, double opts[SBA_OPTSSZ], double
    info[SBA_INFOSZ]);
```

In all cases, the function pointed to by `proj` is assumed to estimate in `xij` the projection in image `j` of the point `i`. Arguments `aj` and `bi` are, respectively, the parameters of the `j`th camera and `i`th point. In other words, `proj` implements the parameterizing function $\mathbf{Q}()$. Similarly, `projac` is assumed to compute in `Aij` and `Bij` the functions $\frac{\partial \mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i)}{\partial \mathbf{a}_j}$ and $\frac{\partial \mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i)}{\partial \mathbf{b}_i}$, that is, the Jacobians with respect to `aj` and `bi` of the projection of point `i` in image `j`. If `projac` is `NULL`, the Jacobians are approximated through the finite differentiation of `proj`. Note that the computation of this finite approximation is more efficient than the one in the case of the expert drivers. This is because the availability of $\mathbf{Q}()$ permits the explicit computation of only the finite differences which actually depend on the differentiating parameters. Both `proj` and `projac` are called only for points `i` which are visible in image `j`. The remaining arguments to the three functions are identical to their homonymous ones in `sba_motstr_levmar_x()`, `sba_mot_levmar_x()`, and `sba_str_levmar_x()`.

6. EXPERIMENTAL RESULTS

As has already been mentioned, `sba` can facilitate the solution of a wide range of reconstruction-related computer vision problems. This section concerns the use of `sba` for Euclidean BA, a task which is a key ingredient for dealing with the problem of camera tracking. Camera tracking refers to using solely visual input for estimating the 3D position and orientation of a freely moving camera. The authors have routinely employed `sba` in this context for dealing with camera tracking problems involving a few thousand 3D points whose image projections depended on a few hundred camera parameters. More specifically, it is assumed that a set of Euclidean 3D points are seen in a number of images acquired by an intrinsically calibrated moving camera. It is also assumed that the image projections of each Euclidean 3D point have been identified and that initial estimates of the 3D point structure and the Euclidean camera matrices have been obtained as described in Lourakis and Argyros [2005a]. The remainder of this section describes the application of `sba` for refining those camera matrix and structure estimates and compares its performance against a variant that employs a more straightforward way for solving the augmented normal equations. Source code demonstrating Euclidean BA using `sba` is included in the `eucsbademo` program distributed with the former.

6.1 Parameterizing Euclidean BA

The employed world coordinate frame is taken to be aligned with that of the camera at the initial location. All subsequent camera motions are defined relative to the initial location, through the combination of a 3D rotation and a 3D translation. A 3D rotation by an angle θ about a unit vector $\mathbf{u} = (u_1, u_2, u_3)^T$ is represented by the quaternion $\mathbf{E} = (\cos(\frac{\theta}{2}),\ u_1 \sin(\frac{\theta}{2}),\ u_2 \sin(\frac{\theta}{2}),\ u_3 \sin(\frac{\theta}{2}))^T$ [Horn 1987; Vicci 2001]. A 3D translation is defined by a vector $\mathbf{t}$. A 3D point is represented by its Euclidean coordinate vector $\mathbf{X}$. Thus, the parameters of each camera j and point i are $\mathbf{a}_j = (\mathbf{E}_j^T, \mathbf{t}_j^T)^T$ and $\mathbf{b}_i = \mathbf{X}_i$, respectively. With the previous definitions, the predicted projection of point i on image j is

$$\mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i) = \mathbf{K}\,(\mathbf{E}_j\ \mathbf{N}_i\ \mathbf{E}_j^{-1}\ +\ \mathbf{t}_j), \tag{29}$$

where $\mathbf{K}$ is the 3×3 intrinsic camera calibration matrix, $\mathbf{E}_j^{-1}$ is the inverse quaternion of $\mathbf{E}_j$, and $\mathbf{N}_i = (0,\ \mathbf{X}_i^T)^T$ is the vector quaternion corresponding to the 3D point $\mathbf{X}_i$. The expression $\mathbf{E}_j\ \mathbf{N}_i\ \mathbf{E}_j^{-1}$ corresponds to point $\mathbf{X}_i$ rotated by an angle θ_j about unit vector $\mathbf{u}_j$, as specified by the quaternion $\mathbf{E}_j$. Routines for evaluating the estimated measurement vector and its Jacobian with respect to all camera and 3D point parameters were implemented. These serve as the `func` and `fjac` arguments of the expert drivers `sba_XXX_levmar_x()`[4]. The computation of the measurement vector's Jacobian relies on a routine for computing the Jacobian of function $\mathbf{Q}()$ in Equation (29), whose code was generated automatically using Maple's symbolic differentiation facilities. Alternatively, the source code for computing the Jacobian of $\mathbf{Q}()$ could have been coded by hand or generated by an automatic differentiation package such as ADOL-C [Griewank et al. 1996]. Additional arguments of `sba_XXX_levmar_x()` assume the following values: `mcon`=1, `cnp`=7, `pnp`=3, `mnp`=2. Notice that setting `mcon` equal to 1 allows the projection matrix of the first camera to be kept constant during bundle adjustment, equal to $\mathbf{K}\,[\mathbf{I}_{3\times3} \mid \mathbf{0}]$.

6.2 Applying `sba` to Euclidean BA

Sample experimental results from a series of camera tracking experiments are presented next. In all experiments, it is assumed that a set of 3D points are seen in a number of images acquired by an intrinsically calibrated moving camera and that the image projections of each 3D point have been identified. Initial estimates of the Euclidean 3D structure and camera motions are then computed using the sequential structure and motion estimation technique described in Lourakis and Argyros [2005a]. With the aid of `sba`, those estimates were then refined through full (i.e., motion and structure) Euclidean BA. No covariance information was considered, that is, $\Sigma_{\mathbf{X}}$ was assumed equal to the identity matrix. Alternatively, point covariance matrices could have been computed as detailed in Brooks et al. [2001]. The set of employed sequences includes the "movi" toy house circular sequence from INRIA's MOVI group, "sagalassos" and "arenberg" from Leuven's VISICS group, "basement" and "house" from Oxford's VGG group, and three sequences acquired by ourselves, namely "maquette," "desk," and "calgrid." The first five are standard sequences, widely used as benchmarks in the reconstruction literature. One frame from each of the employed test sequences is shown in Figure 5. To help the reader visualize the sparseness of the BA problems associated with the sequences at hand, Figure 6 illustrates the sparsity pattern of the 992×992 approximate Hessian matrix (i.e., $\mathbf{J}^T\mathbf{J}$) which corresponds to the "basement" sequence.

Table I illustrates several statistics gathered from the application of Euclidean sparse BA to the eight test sequences. Each row corresponds to a single sequence and columns are as follows: the first column corresponds to the total number of images that were employed in BA. The second column is the total number of 3D points involved in BA. The third column is the total number of motion and structure variables pertaining to the minimization. The fourth column

[4]`XXX` stands for any of `motstr`, `mot` or `str`.

Table I. Execution Statistics for the Application of sba to Euclidean BA
(Total number of images, total number of 3D points, total number of variables, average initial squared reprojection error in pixels, average final squared reprojection error in pixels, total number of objective function/Jacobian evaluations, total number of iterations, elapsed execution time in seconds. Identical values for the user-defined parameters have been used throughout all experiments.)

Sequence	Imgs	Pts	Vars	Init. err.	Fin. err.	Func/jac	Iter.	Time
“movi”	59	1778	5747	5.03	0.3159	20/20	20	3.27
“sagalassos”	26	1709	5309	11.04	1.2703	38/30	30	3.40
“arenberg”	22	1335	4159	1.35	0.5399	27/18	18	2.57
“basement”	11	305	992	0.37	0.2447	37/23	23	0.29
“house”	10	515	1615	1.43	0.2142	28/19	19	0.36
“maquette”	54	5207	15999	2.15	0.1765	30/22	22	7.15
“desk”	46	3422	10588	4.16	1.5760	32/22	22	5.41
“calgrid”	27	722	2355	3.21	0.2297	20/20	20	7.14

corresponds to the average squared reprojection error of the initial reconstruction. The fifth column shows the average squared reprojection error after BA. The sixth column shows the total number of objective function/Jacobian evaluations during BA. The number of iterations needed for convergence is shown in column seven and the last column shows the time (in seconds) elapsed during execution of BA. In all cases, the sparse LM algorithm terminated due to the magnitude of the computed step δ being very small. All experiments were conducted on a 1.8 GHz Intel P4 (512MB RAM, 8KB L1 and 256KB L2 cache) running Linux, `gcc -O3` (v.2.96) and unoptimized BLAS.

As is evident from these results, sba has successfully solved all underlying minimization problems in a short time. For comparison, we have also implemented BA using a dense, general purpose version of the LM algorithm.[5] Note that the dense Jacobians for some of the test problems are extremely large: For example, the “movi” sequence involves 5747 variables and 12,415 image projections. Taking into account that each image projection is a 2D vector and that each double precision real number requires 8 bytes to be stored, the amount of physical memory necessary to store the Jacobian in this case exceeds 1 GB and is thus beyond the capacity of many current desktop computers. Therefore, using dense BA, we have been able to solve only the problems associated with the shortest of the test sequences, namely, those for “basement” and “house,” for which the corresponding execution times were 481.54 and 1960.92 seconds, respectively. Compared to the fractions of a second that were spent by sba for solving those problems, these figures clearly demonstrate the enormous computational gains achieved by the sparse BA implementation.

6.3 Comparison with a Variant Employing Sparse Cholesky Factorization

At this point, a question arising naturally concerns the efficiency of the algorithm of Figure 4 for solving the augmented normal equations in comparison with an approach based on a general purpose, sparse matrix factorization

[5]Our free C/C++ implementation of a dense LM algorithm can be found at `http://www.ics.forth.gr/~lourakis/levmar`.

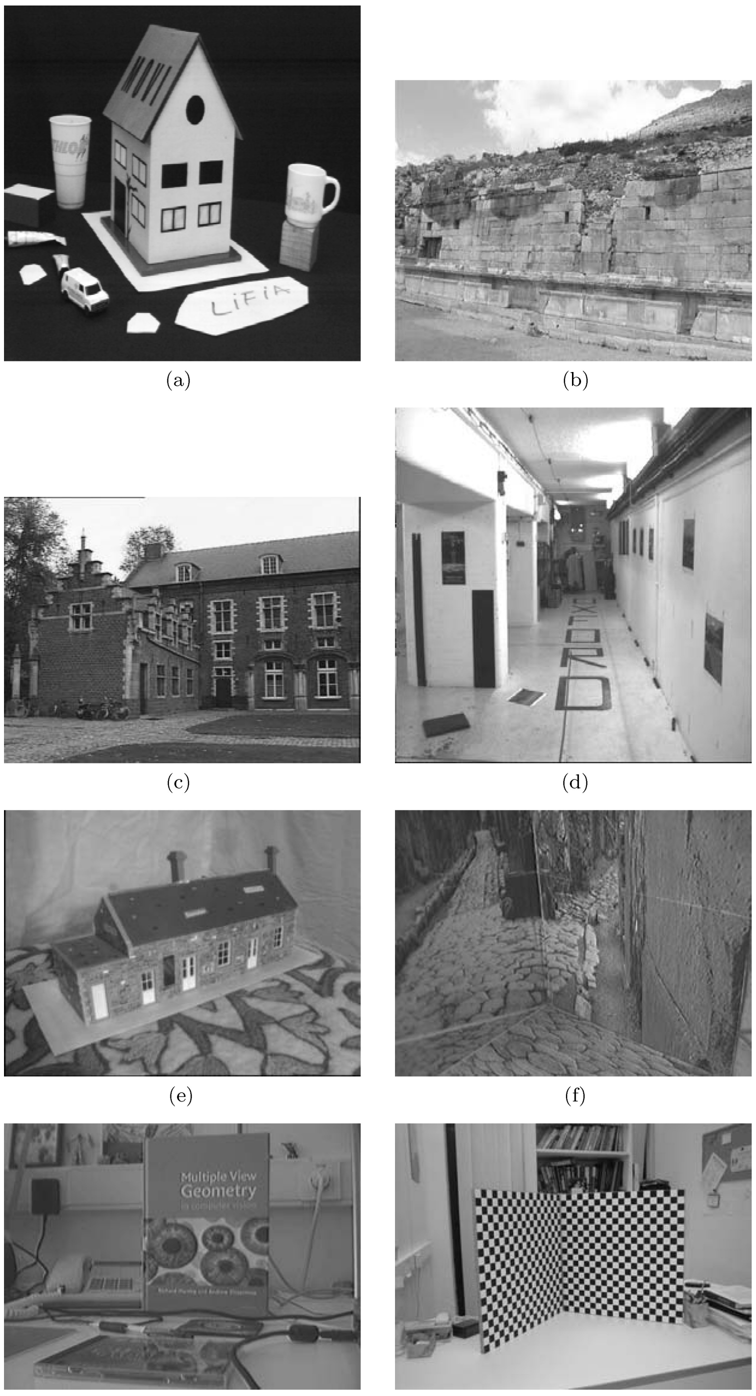

Fig. 5. First frames of the test image sequences employed in Euclidean BA: (a) “movi”; (b) “sagalassos”; (c) “arenberg”; (d) “basement”; (e) “house”; (f) “maquette”; (g) “desk”, and (h) “calgrid.”

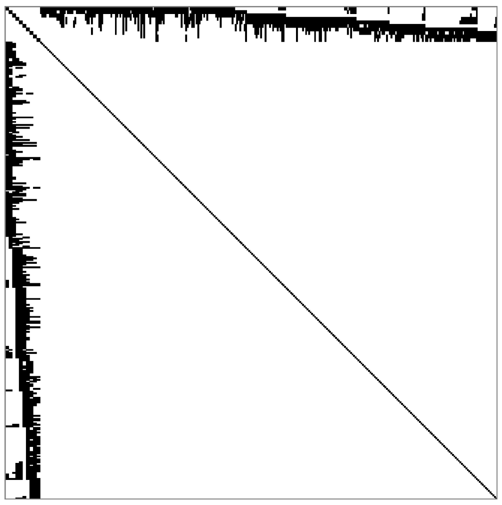

Fig. 6. Sparsity pattern of the approximate 992×992 Hessian corresponding to the "basement" sequence. Black dots correspond to nonzero elements.

algorithm. More specifically, an alternative approach for exploiting the sparseness of the augmented normal equations matrix, would be to factorize it using an appropriate sparse factorization and then solve the augmented normal equations through substitution. Taking into account that the augmented normal equations matrix is symmetric and positive definite, the most appropriate factorization method in this case is Cholesky. To investigate the performance of such an approach, we have implemented a variant of `sba`, which instead of the algorithm of Figure 4, relies on a sparse Cholesky factorization implemented by the `LDL` algorithm [Davis 2005]. `LDL` factorizes a symmetric positive-definite sparse matrix supplied in compressed column storage [Barrett et al. 1994] as $\mathbf{LDL}^T$, where $\mathbf{L}$ is lower triangular and $\mathbf{D}$ diagonal. To ensure good performance, it is crucial that `LDL` be applied to a sparseness-preserving ordering of the original matrix. More formally, letting $\mathbf{N}$ denote the sparse block matrix of Equation (18), `LDL` is employed to factorize the preordered matrix $\mathbf{PNP}^T$. $\mathbf{P}$ is a permutation matrix, which we have chosen to compute with the aid of the `AMD` set of routines [Amestoy et al. 2004]. `AMD` finds $\mathbf{P}$ using a minimum degree ordering algorithm, so that the Cholesky factorization of $\mathbf{PNP}^T$ has fewer nonzero entries (i.e., reduced fill-in) than that of $\mathbf{N}$. Note that, given the $\mathbf{LDL}^T$ factorization of $\mathbf{PNP}^T$, the solution of the linear system $\mathbf{N}\delta = \epsilon$ of Equation (18) can be determined by first solving $\mathbf{Ly} = \mathbf{P}\epsilon$ for $\mathbf{y}$, then $\mathbf{L}^T\mathbf{z} = \mathbf{D}^{-1}\mathbf{y}$ for $\mathbf{z}$, and finally setting $\delta = \mathbf{P}^T\mathbf{z}$.

Table II summarizes the execution statistics gathered by using the `LDL` variant of `sba` to perform Euclidean BA for the same image sequences that were employed in the experiments of Section 6.2. All experiments were conducted on the same Linux system, using the same initial motion and structure estimates as in the experiments of Table I as well as identical user-defined minimization parameters. As can be verified from the figures in the last column, the approach for solving the augmented normal equations that was outlined in Figure 4 is

Table II. Execution Statistics for Euclidean BA Using LDL (Total number of objective function/Jacobian evaluations, total number of iterations, elapsed execution time in seconds. The remaining measures such as number of images, points and variables, average initial squared reprojection error, etc, are as in Table I and are therefore not repeated here.)

Sequence	func/jac	iter.	time
"movi"	23/19	19	8.52
"sagalassos"	42/30	30	9.60
"arenberg"	24/21	21	5.89
"basement"	32/21	21	0.64
"house"	28/19	19	1.04
"maquette"	29/22	22	17.59
"desk"	30/25	25	15.04
"calgrid"	26/20	20	15.71

between 2.2 and 2.9 times faster compared to that based on the LDL factorization, thus confirming the expectation that a custom solver is faster compared to a general purpose one. Therefore, the approach of Figure 4 is preferable for production quality BA. However, when execution time is not the primary concern, the results in Table II indicate the LDL-based approach is a viable and simpler to implement approach for dealing with the augmented normal equations originating from BA.

7. CONCLUSIONS

This article has presented the mathematical theory behind a LM-based sparse bundle adjustment algorithm and has resolved the technical/practical issues pertaining to its implementation in C. The outcome of this work is a generic sparse BA package called sba that has experimentally been demonstrated to be capable of dealing efficiently with very large BA problems. The package can be very useful to researchers working in fields such as computer vision, robotics, image-based graphics, photogrammetry, surveying, geomatics, etc., and therefore it has been made freely available. Indeed, several works employing sba in diverse fields have already been published; a partial list includes Xin et al. [2005], Snavely et al. [2006], Bianchi et al. [2006], Sünderhauf et al. [2005], Templeton et al. [2007], Irschara et al. [2007], and Besnerais et al. [2008]. To the best of our knowledge, sba is the first and currently the only such software package to be offered free of charge.

A possible future enhancement to sba might concern to treat the block matrix **S** as sparse whenever its density is sufficiently low. In such cases, Equation (21) should be solved with a sparse direct solver such as CHOLMOD [Chen et al. 2006] or SuperLU [Demmel et al. 1999]. Another improvement would be to replace the LM algorithm with a more efficient nonlinear least-squares optimization technique such as Powell's dog leg that was proposed in Lourakis and Argyros [2005b].

ACKNOWLEDGMENTS

The authors wish to thank the two anonymous referees whose meticulous reading of earlier versions of the article have contributed to considerable improvements both in the text and sba's code.

REFERENCES

AMESTOY, P., DAVIS, T., AND DUFF, I. 2004. Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm. *ACM Trans. Math. Softw. 30*, 3 (Sep.), 381–388.

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., CROZ, J. D., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' Guide*, 3rd ed. SIAM, Philadelphia, PA.

BARRETT, R., BERRY, M., CHAN, T. F., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND DER VORST, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. SIAM, Philadelphia, PA. `http://www.netlib.org/linalg/html_templates/report.html`.

BEARDSLEY, P., TORR, P., AND ZISSERMAN, A. 1996. 3D model acquisition from extended image sequences. In *Proceedings of the. European Conference on Computer Vision*. Springer-Verlag, Berlin, Germany, 683–695.

BESNERAIS, G. L., SANFOURCHE, M., AND CHAMPAGNAT, F. 2008. Dense height map estimation from oblique aerial image sequences. *Comput. Vis. Image Understand. 109*, 2, 204–225.

BIANCHI, G., JUNG, C., KNOERLEIN, B., SZEKELY, G., AND HARDERS, M. 2006. High-fidelity visuo-haptic interaction with virtual objects in multi-modal AR systems. In *Proceedings of the IEEE/ACM ISMAR'06* (Santa Barbara, CA), 187–196.

BROOKS, M., CHOJNACKI, W., GAWLEY, D., AND VAN DEN HENGEL, A. 2001. What value covariance information in estimating vision parameters? In *Proceedings of the International Conference on Computer Vision*. Vol. I. IEEE Press, Los Alamitos, CA, 302–308.

BROWN, D. 1958. A solution to the general problem of multiple station analytical stereo triangulation. Tech. rep. 43. RCA-MTP. Feb.

CHEN, Y., DAVIS, T., HAGER, W., AND RAJAMANICKAM, S. 2006. Algorithm 8xx: CHOLMOD, supernodal sparse cholesky factorization and update/downdate. Tech. rep. 005. Computer and Information Science and Engineering Dept., University of Florida. Gainesville, FL. Submitted to *ACM Trans. Math. Software*.

CONN, A., GOULD, N., AND TOINT, P. 2000. *Trust Region Methods*. MPS-SIAM Series On Optimization. SIAM, Philadelphia, PA.

DAVIS, T. A. 2005. Algorithm 849: A concise sparse cholesky factorization package. *ACM Trans. Math. Softw. 31*, 4, 587–591.

DEMMEL, J. 1997. *Applied Numerical Linear Algebra*. Titles in Applied Mathematics. SIAM Publications, Philadelphia, PA.

DEMMEL, J., EISENSTAT, S., GILBERT, J., LI, X., AND LIU, J. 1999. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl. 20*, 3, 720–755.

DENNIS, J. 1977. Nonlinear least-squares. In *State of the Art in Numerical Analysis*, D. Jacobs, Ed. Academic Press, New York, NY, 269–312.

DENNIS, J., GAY, D., AND WELSCH, R. 1981. An adaptive nonlinear least-squares algorithm. *ACM Trans. Math. Softw. 7*, 3 (Sep.), 348–368.

DENNIS, J. AND SCHNABEL, R. 1996. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics. SIAM Publications, Philadelphia, PA.

FITZGIBBON, A. AND ZISSERMAN, A. 1998. Automatic camera recovery for closed or open image sequences. In *Proceedings of the European Conference on Computer Vision*. Springer-Verlag, Berlin, Germany, 311–326.

GAY, D. 1990. Usage summary for selected optimization routines. Tech. rep. 153. AT&T Bell Laboratories, Murray Hill, NJ.

GOLUB, G. AND VAN LOAN, C. 1996. *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore, MD.

GRIEWANK, A., JUEDES, D., AND UTKE, J. 1996. ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw. 22*, 2, 131–167.

HARTLEY, R. 1993. Euclidean reconstruction from uncalibrated views. In *Applications of Invariance in Computer Vision*, J. Mundy and A. Zisserman, Eds. Lecture Notes in Computer Science, Vol. 825. Springer-Verlag, Berlin, Germany, 237–256.

HARTLEY, R. AND ZISSERMAN, A. 2000. *Multiple View Geometry in Computer Vision*, 1st ed. Cambridge University Press, Cambridge, MA.

HIEBERT, K. 1981. An evaluation of mathematical software that solves nonlinear least squares problems. *ACM Trans. Math. Softw. 7*, 1 (Mar.), 1–16.

HORN, B. 1987. Closed-form solution of absolute orientation using unit quaternions. *J. Opt. Soc. Amer. A 4*, 4, 629–642.

IRSCHARA, A., ZACH, C., AND BISCHOF, H. 2007. Towards wiki-based dense city modeling. In *Proceedings of the International Conference on Computer Vision*. IEEE Press, Los Alamitos, CA, 1–8.

KELLEY, C. 1999. *Iterative Methods for Optimization*. SIAM Publications, Philadelphia.

LAMPTON, M. 1997. Damping-undamping strategies for the Levenberg-Marquardt nonlinear least-squares method. *Comput. Phys. J. 11*, 1 (Jan./Feb.), 110–115.

LEVENBERG, K. 1944. A method for the solution of certain non-linear problems in least squares. *Quart. Appl. Math. 2*, 2 (Jul.), 164–168.

LOURAKIS, M. AND ARGYROS, A. 2005a. Efficient, causal camera tracking in unprepared environments. *Comput. Vis. and Image Understand. 99*, 2 (Aug.), 259–290.

LOURAKIS, M. AND ARGYROS, A. 2005b. Is Levenberg-Marquardt the most efficient optimization algorithm for implementing bundle adjustment? In *Proceedings of the International Conference on Computer Vision*. IEEE Press, Los Alamitos, CA, 1526–1531.

LOWE, D. 2004. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vis. 60*, 2 (Nov.), 91–110.

LU, C.-P., HAGER, G., AND MJOLSNESS, E. 2000. Fast and globally convergent pose estimation from video images. *IEEE Trans. Patt. Anal. Mach. Intell. 22*, 6 (Jun.), 610–622.

MADSEN, K., NIELSEN, H., AND TINGLEFF, O. 2004. Methods for non-linear least squares problems. Technical University of Denmark. Lecture notes. `http://www.imm.dtu.dk/pubdb/views/edoc_download.php/3215/pdf/imm3215.pdf`.

MARQUARDT, D. 1963. An algorithm for the least-squares estimation of nonlinear parameters. *SIAM J. Appl. Math. 11*, 2 (Jun.), 431–441.

MORÉ, J. 1977. The Levenberg Marquardt algorithm: implementation and theory. In *Numerical Analysis*, G. Watson, Ed. Lecture Notes in Mathematics, vol. 630. Springer Verlag, Berlin, Germany, 105–116.

MORÉ, J., GARBOW, B., AND HILLSTROM, K. 1980. User guide for MINPACK-1. Tech. rep. ANL-80-74. Argonne National Laboratory, Argonne, IL.

MORÉ, J. AND SORENSEN, D. 1983. Computing a trust region step. *SIAM J. Sci. Statist. Comput. 4*, 553–572.

MUNDY, J. AND ZISSERMAN, A. 1992. Projective geometry for machine vision. In *Geometric Invariance in Computer Vision*, J. Mundy and A. Zisserman, Eds. MIT Press, Cambridge, MA, 463–519.

NIELSEN, H. 1999. Damping parameter in Marquardt's method. Tech. rep. IMM-REP-1999-05. Technical University of Denmark, Aarhus, Denmark. `http://www.imm.dtu.dk/~hbn`.

NOCEDAL, J. AND WRIGHT, S. 1999. *Numerical Optimization*. Springer, New York, NY.

POLLEFEYS, M., GOOL, L. V., VERGAUWEN, M., VERBIEST, F., CORNELIS, K., TOPS, J., AND KOCH, R. 2004. Visual modeling with a hand-held camera. *Int. J. Comput. Vis. 59*, 3 (Sep./Oct.), 207–232.

PRASOLOV, V. 1994. *Problems and Theorems in Linear Algebra*. American Mathematical Society, Providence, RI.

SEMPLE, J. AND KNEEBONE, G. 1952. *Algebraic Projective Geometry*. Oxford University Press, Oxford, U.K.

SHI, J. AND TOMASI, C. 1994. Good features to track. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*. IEEE Press, Los Alamitos, CA, 593–600.

SHUM, H.-Y., KE, Q., AND ZHANG, Z. 1999. Efficient bundle adjustment with virtual key frames: a hierarchical approach to multi-frame structure from motion. In *Proceedings of the International*

Conference on Computer Vision and Pattern Recognition. Vol. 2. IEEE Press, Los Alamitos, CA, 538–543.

SLAMA, C. 1980. *Manual of Photogrammetry*, ed. American Society of Photogrammetry, Falls Church, VA.

SNAVELY, N., SEITZ, S., AND SZELISKI, R. 2006. Photo tourism: Exploring photo collections in 3D. *ACM Trans. Graphics (SIGGRAPH Proceedings) 25*, 3, 835–846.

SÜNDERHAUF, N., KONOLIGE, K., LACROIX, S., AND PROTZEL, P. 2005. Visual odometry using sparse bundle adjustment on an autonomous outdoor vehicle. In *Autonome Mobile Systeme 2005*, P. Levi, M. Schanz, R. Lafrenz, and V. Avrutin, Eds. Informatik Aktuell. Springer Verlag, Berlin, Germany, 157–163.

SZELISKI, R. AND KANG, S. 1994. Recovering 3D shape and motion from image streams using nonlinear least squares. *J. Vis. Comm. Im. Repr. 5*, 1, 10–28.

TEMPLETON, T., SHIM, D., GEYER, C., AND SASTRY, S. 2007. Autonomous vision-based landing and terrain mapping using an MPC-controlled unmanned rotorcraft. In *Proceedings of the IEEE ICRA'07*. 1349–1356.

TRIGGS, B., MCLAUCHLAN, P., HARTLEY, R., AND FITZGIBBON, A. 1999. Bundle adjustment—a modern synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*. 298–372.

VICCI, L. 2001. Quaternions and rotations in 3-space: The algebra and its geometric interpretation. Tech. rep. TR01-014. Dept. of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, NC.

XIN, L., WANG, Q., TAO, J., TANG, X., TAN, T., AND SHUM, H. 2005. Automatic 3D face modeling from video. In *Proceedings of the International Conference on Computer Vision*. Vol. 2. IEEE Press, Los Alamitos, CA, 1193–1199.

ZHANG, Z., DERICHE, R., FAUGERAS, O., AND LUONG, Q.-T. 1995. A robust technique for matching two uncalibrated images through the recovery of the unknown epipolar geometry. *AI J. 78*, 87–119. (Detailed version in INRIA Tech.rep. RR-2273. INRIA Rocquencourt, France.)

ZHANG, Z. AND SHAN, Y. 2003. Incremental motion estimation through modified bundle adjustment. In *Proceedings of the International Conference on Image Processing*. Vol. 2. 343–346.

Received May 2005; revised February 2006, February 2007; accepted May 2008