FORTH-ICS / TR-340 August 2004

The Design and Implementation of a Generic Sparse Bundle Adjustment Software Package Based on the Levenberg-Marquardt Algorithm†

Manolis I.A. Lourakis and Antonis A. Argyros

Bundle adjustment using the Levenberg-Marquardt minimization algorithm is almost invariably used as the last step of every feature-based structure and motion estimation vision algorithm to obtain optimal 3D structure and viewing parameter estimates. However, due to the large number of unknowns contributing to the minimized reprojection error, a general purpose implementation of the Levenberg-Marquardt algorithm incurs high computational costs when applied to the problem of bundle adjustment. Fortunately, the lack of interaction among parameters for different 3D points and cameras in multiple view reconstruction results in the underlying normal equations exhibiting a sparse block structure, which can be exploited to gain considerable computational benefits. This paper presents the design and explains the use of `sba`, a publicly available C/C++ software package for generic bundle adjustment based on the sparse Levenberg-Marquardt algorithm.

The most recent revision of this document will always be found at
http://www.ics.forth.gr/~lourakis/sba.
This version was last edited on December 30, 2004

† This work was partially supported by the EU IST-2001-34545 project LifePlus.

The Design and Implementation of a Generic Sparse Bundle Adjustment Software Package Based on the Levenberg-Marquardt Algorithm

Manolis I.A. Lourakis and Antonis A. Argyros

Computational Vision and Robotics Laboratory
Institute of Computer Science (ICS)
Foundation for Research and Technology — Hellas (FORTH)
Science and Technology Park, Heraklio, Crete
P.O. Box 1385, GR-711-10 Greece

Web: http://www.ics.forth.gr/cvrl – http://www.ics.forth.gr/~lourakis/sba
E-mail: {lourakis | argyros}@ics.forth.gr
Tel: +30 2810 391716, Fax: +30 2810 391601

Technical Report FORTH-ICS / TR-340 — August 2004

©Copyright 2004 by FORTH

Bundle adjustment using the Levenberg-Marquardt minimization algorithm is almost invariably used as the last step of every feature-based structure and motion estimation vision algorithm to obtain optimal 3D structure and viewing parameter estimates. However, due to the large number of unknowns contributing to the minimized reprojection error, a general purpose implementation of the Levenberg-Marquardt algorithm incurs high computational costs when applied to the problem of bundle adjustment. Fortunately, the lack of interaction among parameters for different 3D points and cameras in multiple view reconstruction results in the underlying normal equations exhibiting a sparse block structure, which can be exploited to gain considerable computational benefits. This paper presents the design and explains the use of `sba`, a publicly available C/C++ software package for generic bundle adjustment based on the sparse Levenberg-Marquardt algorithm.

1 Introduction

Bundle Adjustment (BA) is almost invariably used as the last step of every feature-based multiview structure and motion estimation algorithm; see, for example, [10, 3, 6, 21, 27]. BA was originally conceived in the field of photogrammetry [24] and has increasingly been used by vision researchers during the last decade. An excellent overview of its application to vision-based reconstruction is given in [25]. BA is a technique for simultaneously refining 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 [25]: 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 centre, which are adjusted optimally with respect to both structure and viewing parameters.

BA amounts to minimizing the reprojection error between the observed and predicted image points, which is expressed as the sum of squares of a number of non-linear real-valued functions. Thus, the minimization is achieved using non-linear least squares algorithms [4, 18], of which the Levenberg-Marquardt (LM) has proven to be the most successful due to its use of an effective damping strategy that lends it the ability to converge promptly from a wide range of initial guesses [11]. 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 parameters, it is clear that general purpose implementations of the LM algorithm, such as MINPACK's LMDER [19] or PORT3's DN2G [1] [5] routines for example, are computationally very demanding when employed to minimize functions depending on many parameters. 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.

This paper presents the design and implementation in ANSI C of `sba`, a generic sparse BA package which is distributed under the terms of the GNU General Public License[2]. `sba` is also usable from C++ and is generic in the sense that it grants the user full control over the choice of 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

[1]By keeping the terms due to the second derivative in the Hessian of $\epsilon^T\epsilon$ (see section 2), 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.

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

as arbitrary projective cameras, partially or fully intrinsically calibrated cameras, exterior orientation (i.e. pose) estimation from fixed 3D points, refinement of intrinsic calibration parameters, etc. The authors have successfully employed `sba` for dealing with the problem of camera tracking [14] in sequences involving a few thousands 3D points whose image projections depended on a few hundreds camera parameters. The `sba` package can be downloaded from `http://www.ics.forth.gr/~lourakis/sba`. The rest of the paper is organized as follows. Section 2 briefly explains the conventional, dense LM algorithm for solving non-linear least squares minimization problems. Section 3 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 4. A sample use case of `sba` is presented in section 5 and the paper is concluded with a brief discussion in section 6.

2 The Levenberg-Marquardt Algorithm

The LM algorithm is an iterative technique that locates the minimum of a multivariate function that is expressed as the sum of squares of non-linear real-valued functions [13, 17]. It has become a standard technique for non-linear least-squares problems, widely adopted in the field of computer vision. LM can be thought of as a combination of steepest descent and the Gauss-Newton method. When the current solution is far from the correct one, the algorithm behaves like a steepest descent method: slow, but guaranteed to converge. When the current solution is close to the correct solution, it becomes a Gauss-Newton method. For completeness purposes, a short description of the LM algorithm based on the material in [16] is supplied next. Note, however, that a detailed analysis of the LM algorithm is beyond the scope of this paper and the interested reader is referred to [16, 20, 22] for more comprehensive treatments.

In the following, vectors and arrays appear in boldface and ^{T} is used to denote transposition. Also, $||.||$ and $||.||_\infty$ denote the 2 and infinity norms respectively. 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, i.e. minimizes the squared distance $\epsilon^T\epsilon$ with $\epsilon = \mathbf{x} - \hat{\mathbf{x}}$. The basis of the LM algorithm is a linear approximation to f in the neighborhood of $\mathbf{p}$. For a small $||\delta_\mathbf{p}||$, a Taylor series expansion leads to the approximation

$$f(\mathbf{p} + \delta_\mathbf{p}) \approx f(\mathbf{p}) + \mathbf{J}\delta_\mathbf{p}, \tag{1}$$

where $\mathbf{J}$ is the Jacobian matrix $\frac{\partial f(\mathbf{p})}{\partial \mathbf{p}}$. Like all non-linear optimization methods, LM is iterative: Initiated at the starting point $\mathbf{p}_0$, the method produces a series of vectors $\mathbf{p}_1, \mathbf{p}_2, \ldots$, that converge towards a local minimizer $\mathbf{p}^+$ for f. Hence, at each step, it is required to find the $\delta_\mathbf{p}$ 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* [7]:

$$\mathbf{J}^T\mathbf{J}\delta_{\mathbf{p}} = \mathbf{J}^T\epsilon. \tag{2}$$

The LM method actually solves a slight variation of Eq. (2), known as the *augmented normal equations*

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

where the off-diagonal elements of $\mathbf{N}$ are identical to the corresponding elements of $\mathbf{J}^T\mathbf{J}$ and the diagonal elements are given by $\mathbf{N}_{ii} = \mu + \left[\mathbf{J}^T\mathbf{J}\right]_{ii}$ for some $\mu > 0$. 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 Eq. (3) leads to a reduction in the error ϵ, 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 error is found. The process of repeatedly solving Eq. (3) 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 ϵ. If the damping is set to a large value, matrix $\mathbf{N}$ in Eq. (3) is nearly diagonal and the LM update step $\delta_{\mathbf{p}}$ is near the steepest descent direction. Moreover, the magnitude of $\delta_{\mathbf{p}}$ is reduced in this case. Damping also handles situations where the Jacobian is rank deficient and $\mathbf{J}^T\mathbf{J}$ is therefore singular [12]. In this way, LM can defensively navigate a region of the parameter space in which the model is highly nonlinear. If the damping is small, the LM step approximates the exact quadratic step appropriate for a fully linear problem. LM is adaptive because it controls its own damping: it raises the damping if a step fails to reduce ϵ; otherwise it reduces the damping. In this way LM is capable to alternate between a slow descent approach when being far from the minimum and a fast convergence when being at the minimum's neighborhood [12]. The LM algorithm terminates when at least one of the following conditions is met:

- The magnitude of the gradient of $-\ \epsilon^T\epsilon$, i.e. $\mathbf{J}^T\epsilon$ in the right hand side of Eq. (2), drops below a threshold ε_1
- The relative change in the magnitude of $\delta_{\mathbf{p}}$ drops below a threshold ε_2
- A maximum number of iterations k_{max} is completed

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

$$\mathbf{J}^T\mathbf{\Sigma_x}^{-1}\mathbf{J}\delta_{\mathbf{p}} = \mathbf{J}^T\mathbf{\Sigma_x}^{-1}\epsilon. \tag{4}$$

The rest of the algorithm remains unchanged. The complete LM algorithm is shown in pseudocode in Fig. 1. It is derived by slight modification of algorithm 3.16 in page 27 of [16]; more details regarding the LM algorithm can be found there. Indicative values for the user-defined parameters are $\tau = 10^{-3}$, $\varepsilon_1 = \varepsilon_2 = 10^{-15}$, $k_{max} = 100$. A free C/C++ implementation of this dense LM algorithm can be found at `http://www.ics.forth.gr/~lourakis/levmar`.

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 ($||\delta_\mathbf{p}|| \leq \varepsilon_2||\mathbf{p}||$)
            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 $\rho > 0$
                $\mathbf{p} = \mathbf{p}_{new}$;
                $\mathbf{A} := \mathbf{J}^T\mathbf{J}$; $\epsilon_\mathbf{p} := \mathbf{x} - f(\mathbf{p})$; $\mathbf{g} := \mathbf{J}^T\epsilon_\mathbf{p}$;
                stop:=($||\mathbf{g}||_\infty \leq \varepsilon_1$);
                $\mu := \mu * \max(\frac{1}{3}, 1 - (2\rho - 1)^3)$; $\nu := 2$;
            else
                $\mu := \mu * \nu$; $\nu := 2 * \nu$;
            endif
        endif
    until ($\rho > 0$) or (stop)
endwhile

Figure 1: Levenberg-Marquardt non-linear least squares algorithm; see text and [16, 20] for details. The reason for enclosing a statement in a rectangular box will be explained in section 3.

3 Sparse Bundle Adjustment

This section shows how can a sparse variant of the LM algorithm presented in section 2 be developed to deal efficiently with the problem of bundle adjustment. The developments and notation that follow are to a large extent based on the presentation regarding sparse bundle adjustment in [9]. To begin, assume that n 3D points are seen in m views and let $\mathbf{x}_{ij}$ be the projection of the i-th point on image j. Bundle adjustment amounts to 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 later 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

$$\min_{\mathbf{a}_j,\mathbf{b}_i} \sum_{i=1}^{n} \sum_{j=1}^{m} d(\mathbf{Q}(\mathbf{a}_j,\mathbf{b}_i),\ \mathbf{x}_{ij})^2, \tag{5}$$

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 inhomogeneous image points represented by $\mathbf{x}$ and $\mathbf{y}$. It is clear from (5) that BA is by definition tolerant to missing image projections and, in contrast to algebraic approaches, minimizes a physically meaningful criterion. Observe that through $\mathbf{Q}()$, the definition in (5) 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 (5) equals $m\kappa + n\lambda$ and is therefore large even for moderately sized BA problems.

BA can be cast as a non-linear 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 Eq. (5), 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:

$$\mathbf{X} = (\mathbf{x}_{11}{}^T, \ldots, \mathbf{x}_{1m}{}^T,\ \mathbf{x}_{21}{}^T, \ldots, \mathbf{x}_{2m}{}^T,\ \ldots,\ \mathbf{x}_{n1}{}^T, \ldots, \mathbf{x}_{nm}{}^T)^T. \tag{6}$$

Let $\mathbf{P}_0$ be an initial parameter estimate and $\mathbf{\Sigma_X}$ the covariance matrix corresponding to the measured vector $\mathbf{X}$ (in the absence of any further knowledge, it is assumed that $\mathbf{\Sigma_X}$ is 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

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

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

Thus, BA corresponds to minimizing the squared Mahalanobis distance $\epsilon^T \mathbf{\Sigma_X}^{-1} \epsilon$, $\epsilon = \mathbf{X} - \hat{\mathbf{X}}$ over $\mathbf{P}$. Evidently, this minimization problem can be solved by using the LM non-linear least squares algorithm to iteratively solve the weighted normal equations

$$\mathbf{J}^T \mathbf{\Sigma}_{\mathbf{X}}^{-1} \mathbf{J} \delta = \mathbf{J}^T \mathbf{\Sigma}_{\mathbf{X}}^{-1} \epsilon, \tag{8}$$

where $\mathbf{J}$ is the Jacobian of f and δ is the sought update to the parameter vector $\mathbf{P}$. As will shortly be demonstrated, the normal equations in Eq. (8) 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.

Taking into account the notation for the derivatives introduced in the previous paragraph, the Jacobian $\mathbf{J}$ is given by

$$\frac{\partial \mathbf{X}}{\partial \mathbf{P}} = \begin{pmatrix}
\mathbf{A}_{11} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{11} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{A}_{12} & \mathbf{0} & \mathbf{B}_{12} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{A}_{13} & \mathbf{B}_{13} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{A}_{21} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{21} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{A}_{22} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{22} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{A}_{23} & \mathbf{0} & \mathbf{B}_{23} & \mathbf{0} & \mathbf{0} \\
\mathbf{A}_{31} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{31} & \mathbf{0} \\
\mathbf{0} & \mathbf{A}_{32} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{32} & \mathbf{0} \\
\mathbf{0} & \mathbf{0} & \mathbf{A}_{33} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{33} & \mathbf{0} \\
\mathbf{A}_{41} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{41} \\
\mathbf{0} & \mathbf{A}_{42} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{42} \\
\mathbf{0} & \mathbf{0} & \mathbf{A}_{43} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{B}_{43}
\end{pmatrix}. \tag{9}$$

Eq. (9) 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

$$\mathbf{\Sigma}_{\mathbf{X}} = diag(\mathbf{\Sigma}_{\mathbf{x}_{11}},\ \mathbf{\Sigma}_{\mathbf{x}_{12}},\ \mathbf{\Sigma}_{\mathbf{x}_{13}},\ \mathbf{\Sigma}_{\mathbf{x}_{21}},\ \mathbf{\Sigma}_{\mathbf{x}_{22}},\ \mathbf{\Sigma}_{\mathbf{x}_{23}},\ \mathbf{\Sigma}_{\mathbf{x}_{31}},\ \mathbf{\Sigma}_{\mathbf{x}_{32}},\ \mathbf{\Sigma}_{\mathbf{x}_{33}},\ \mathbf{\Sigma}_{\mathbf{x}_{41}},\ \mathbf{\Sigma}_{\mathbf{x}_{42}},\ \mathbf{\Sigma}_{\mathbf{x}_{43}}). \tag{10}$$

Substituting $\mathbf{J}$ and $\mathbf{\Sigma}_{\mathbf{X}}^{-1}$ from Eqs. (9) and (10), the matrix product in the left hand side of Eq. (8) becomes

$$\begin{pmatrix}
\sum_{i=1}^{4} \mathbf{A}_{i1}{}^T \mathbf{\Sigma}_{\mathbf{x}_{i1}}^{-1} \mathbf{A}_{i1} & \mathbf{0} & \mathbf{0} & \mathbf{A}_{11}{}^T \mathbf{\Sigma}_{\mathbf{x}_{11}}^{-1} \mathbf{B}_{11} & \mathbf{A}_{21}{}^T \mathbf{\Sigma}_{\mathbf{x}_{21}}^{-1} \mathbf{B}_{21} & \mathbf{A}_{31}{}^T \mathbf{\Sigma}_{\mathbf{x}_{31}}^{-1} \mathbf{B}_{31} & \mathbf{A}_{41}{}^T \mathbf{\Sigma}_{\mathbf{x}_{41}}^{-1} \mathbf{B}_{41} \\
\mathbf{0} & \sum_{i=1}^{4} \mathbf{A}_{i2}{}^T \mathbf{\Sigma}_{\mathbf{x}_{i2}}^{-1} \mathbf{A}_{i2} & \mathbf{0} & \mathbf{A}_{12}{}^T \mathbf{\Sigma}_{\mathbf{x}_{12}}^{-1} \mathbf{B}_{12} & \mathbf{A}_{22}{}^T \mathbf{\Sigma}_{\mathbf{x}_{22}}^{-1} \mathbf{B}_{22} & \mathbf{A}_{32}{}^T \mathbf{\Sigma}_{\mathbf{x}_{32}}^{-1} \mathbf{B}_{32} & \mathbf{A}_{42}{}^T \mathbf{\Sigma}_{\mathbf{x}_{42}}^{-1} \mathbf{B}_{42} \\
\mathbf{0} & \mathbf{0} & \sum_{i=1}^{4} \mathbf{A}_{i3}{}^T \mathbf{\Sigma}_{\mathbf{x}_{i3}}^{-1} \mathbf{A}_{i3} & \mathbf{A}_{13}{}^T \mathbf{\Sigma}_{\mathbf{x}_{13}}^{-1} \mathbf{B}_{13} & \mathbf{A}_{23}{}^T \mathbf{\Sigma}_{\mathbf{x}_{23}}^{-1} \mathbf{B}_{23} & \mathbf{A}_{33}{}^T \mathbf{\Sigma}_{\mathbf{x}_{33}}^{-1} \mathbf{B}_{33} & \mathbf{A}_{43}{}^T \mathbf{\Sigma}_{\mathbf{x}_{43}}^{-1} \mathbf{B}_{43} \\
\mathbf{B}_{11}{}^T \mathbf{\Sigma}_{\mathbf{x}_{11}}^{-1} \mathbf{A}_{11} & \mathbf{B}_{12}{}^T \mathbf{\Sigma}_{\mathbf{x}_{12}}^{-1} \mathbf{A}_{12} & \mathbf{B}_{13}{}^T \mathbf{\Sigma}_{\mathbf{x}_{13}}^{-1} \mathbf{A}_{13} & \sum_{j=1}^{3} \mathbf{B}_{1j}{}^T \mathbf{\Sigma}_{\mathbf{x}_{1j}}^{-1} \mathbf{B}_{1j} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{B}_{21}{}^T \mathbf{\Sigma}_{\mathbf{x}_{21}}^{-1} \mathbf{A}_{21} & \mathbf{B}_{22}{}^T \mathbf{\Sigma}_{\mathbf{x}_{22}}^{-1} \mathbf{A}_{22} & \mathbf{B}_{23}{}^T \mathbf{\Sigma}_{\mathbf{x}_{23}}^{-1} \mathbf{A}_{23} & \mathbf{0} & \sum_{j=1}^{3} \mathbf{B}_{2j}{}^T \mathbf{\Sigma}_{\mathbf{x}_{2j}}^{-1} \mathbf{B}_{2j} & \mathbf{0} & \mathbf{0} \\
\mathbf{B}_{31}{}^T \mathbf{\Sigma}_{\mathbf{x}_{31}}^{-1} \mathbf{A}_{31} & \mathbf{B}_{32}{}^T \mathbf{\Sigma}_{\mathbf{x}_{32}}^{-1} \mathbf{A}_{32} & \mathbf{B}_{33}{}^T \mathbf{\Sigma}_{\mathbf{x}_{33}}^{-1} \mathbf{A}_{33} & \mathbf{0} & \mathbf{0} & \sum_{j=1}^{3} \mathbf{B}_{3j}{}^T \mathbf{\Sigma}_{\mathbf{x}_{3j}}^{-1} \mathbf{B}_{3j} & \mathbf{0} \\
\mathbf{B}_{41}{}^T \mathbf{\Sigma}_{\mathbf{x}_{41}}^{-1} \mathbf{A}_{41} & \mathbf{B}_{42}{}^T \mathbf{\Sigma}_{\mathbf{x}_{42}}^{-1} \mathbf{A}_{42} & \mathbf{B}_{43}{}^T \mathbf{\Sigma}_{\mathbf{x}_{43}}^{-1} \mathbf{A}_{43} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \sum_{j=1}^{3} \mathbf{B}_{4j}{}^T \mathbf{\Sigma}_{\mathbf{x}_{4j}}^{-1} \mathbf{B}_{4j}
\end{pmatrix}. \tag{11}$$

Denoting $\sum_{i=1}^{4} \mathbf{A}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{A}_{ij}$, $\sum_{j=1}^{3} \mathbf{B}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{B}_{ij}$ and $\mathbf{A}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{B}_{ij}$ by $\mathbf{U}_j$, $\mathbf{V}_i$ and $\mathbf{W}_{ij}$ respectively, the above matrix can be written in shorthand form as

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

Also, using Eqs. (9) and (10), the right hand side of Eq. (8) can be expanded as

$$\begin{pmatrix}
\sum_{i=1}^{4} \mathbf{A}_{i1}{}^T \mathbf{\Sigma}_{\mathbf{x}_{i1}}^{-1} \epsilon_{i1} \\
\sum_{i=1}^{4} \mathbf{A}_{i2}{}^T \mathbf{\Sigma}_{\mathbf{x}_{i2}}^{-1} \epsilon_{i2} \\
\sum_{i=1}^{4} \mathbf{A}_{i3}{}^T \mathbf{\Sigma}_{\mathbf{x}_{i3}}^{-1} \epsilon_{i3} \\
\sum_{j=1}^{3} \mathbf{B}_{1j}{}^T \mathbf{\Sigma}_{\mathbf{x}_{1j}}^{-1} \epsilon_{1j} \\
\sum_{j=1}^{3} \mathbf{B}_{2j}{}^T \mathbf{\Sigma}_{\mathbf{x}_{2j}}^{-1} \epsilon_{2j} \\
\sum_{j=1}^{3} \mathbf{B}_{3j}{}^T \mathbf{\Sigma}_{\mathbf{x}_{3j}}^{-1} \epsilon_{3j} \\
\sum_{j=1}^{3} \mathbf{B}_{4j}{}^T \mathbf{\Sigma}_{\mathbf{x}_{4j}}^{-1} \epsilon_{4j}
\end{pmatrix}. \tag{13}$$

Denoting $\sum_{i=1}^{4} \mathbf{A}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \epsilon_{ij}$ and $\sum_{j=1}^{3} \mathbf{B}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \epsilon_{ij}$ by $\epsilon_{\mathbf{a}_j}$ and $\epsilon_{\mathbf{b}_i}$ respectively, vector (13) can be abbreviated to

$$(\epsilon_{\mathbf{a}_1}{}^T, \epsilon_{\mathbf{a}_2}{}^T, \epsilon_{\mathbf{a}_3}{}^T, \epsilon_{\mathbf{b}_1}{}^T, \epsilon_{\mathbf{b}_2}{}^T, \epsilon_{\mathbf{b}_3}{}^T, \epsilon_{\mathbf{b}_4}{}^T)^T. \tag{14}$$

Substituting the expressions for $\mathbf{J}^T \mathbf{\Sigma}_{\mathbf{X}}^{-1} \epsilon$ and $\mathbf{J}^T \mathbf{\Sigma}_{\mathbf{X}}^{-1} \epsilon$ from (12) and (14), the normal equations (8) become

$$\begin{pmatrix} \mathbf{U}_1 & \mathbf{0} & \mathbf{0} & \mathbf{W}_{11} & \mathbf{W}_{21} & \mathbf{W}_{31} & \mathbf{W}_{41} \\ \mathbf{0} & \mathbf{U}_2 & \mathbf{0} & \mathbf{W}_{12} & \mathbf{W}_{22} & \mathbf{W}_{32} & \mathbf{W}_{42} \\ \mathbf{0} & \mathbf{0} & \mathbf{U}_3 & \mathbf{W}_{13} & \mathbf{W}_{23} & \mathbf{W}_{33} & \mathbf{W}_{43} \\ \mathbf{W}_{11}{}^T & \mathbf{W}_{12}{}^T & \mathbf{W}_{13}{}^T & \mathbf{V}_1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{W}_{21}{}^T & \mathbf{W}_{22}{}^T & \mathbf{W}_{23}{}^T & \mathbf{0} & \mathbf{V}_2 & \mathbf{0} & \mathbf{0} \\ \mathbf{W}_{31}{}^T & \mathbf{W}_{32}{}^T & \mathbf{W}_{33}{}^T & \mathbf{0} & \mathbf{0} & \mathbf{V}_3 & \mathbf{0} \\ \mathbf{W}_{41}{}^T & \mathbf{W}_{42}{}^T & \mathbf{W}_{43}{}^T & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{V}_4 \end{pmatrix} \begin{pmatrix} \delta_{\mathbf{a}_1} \\ \delta_{\mathbf{a}_2} \\ \delta_{\mathbf{a}_3} \\ \delta_{\mathbf{b}_1} \\ \delta_{\mathbf{b}_2} \\ \delta_{\mathbf{b}_3} \\ \delta_{\mathbf{b}_4} \end{pmatrix} = \begin{pmatrix} \epsilon_{\mathbf{a}_1} \\ \epsilon_{\mathbf{a}_2} \\ \epsilon_{\mathbf{a}_3} \\ \epsilon_{\mathbf{b}_1} \\ \epsilon_{\mathbf{b}_2} \\ \epsilon_{\mathbf{b}_3} \\ \epsilon_{\mathbf{b}_4} \end{pmatrix}. \tag{15}$$

Denoting

$$\mathbf{U}^* = \begin{pmatrix} \mathbf{U}_1^* & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2^* & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{U}_3^* \end{pmatrix}, \mathbf{V}^* = \begin{pmatrix} \mathbf{V}_1^* & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_2^* & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{V}_3^* & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{V}_4^* \end{pmatrix}, \mathbf{W} = \begin{pmatrix} \mathbf{W}_{11} & \mathbf{W}_{21} & \mathbf{W}_{31} & \mathbf{W}_{41} \\ \mathbf{W}_{12} & \mathbf{W}_{22} & \mathbf{W}_{32} & \mathbf{W}_{42} \\ \mathbf{W}_{13} & \mathbf{W}_{23} & \mathbf{W}_{33} & \mathbf{W}_{43} \end{pmatrix}, \tag{16}$$

where * designates the augmentation of diagonal elements, allows the augmented normal equations to be further compacted to

$$\begin{pmatrix} \mathbf{U}^* & \mathbf{W} \\ \mathbf{W}^T & \mathbf{V}^* \end{pmatrix} \begin{pmatrix} \delta_{\mathbf{a}} \\ \delta_{\mathbf{b}} \end{pmatrix} = \begin{pmatrix} \epsilon_{\mathbf{a}} \\ \epsilon_{\mathbf{b}} \end{pmatrix}. \tag{17}$$

Left multiplication of Eq. (17) by the block matrix

$$\begin{pmatrix} \mathbf{I} & -\mathbf{W}\ \mathbf{V}^{*-1} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \tag{18}$$

results in

$$\begin{pmatrix} \mathbf{U}^* - \mathbf{W}\ \mathbf{V}^{*-1}\ \mathbf{W}^T & \mathbf{0} \\ \mathbf{W}^T & \mathbf{V}^* \end{pmatrix} \begin{pmatrix} \delta_{\mathbf{a}} \\ \delta_{\mathbf{b}} \end{pmatrix} = \begin{pmatrix} \epsilon_{\mathbf{a}} - \mathbf{W}\ \mathbf{V}^{*-1}\ \epsilon_{\mathbf{b}} \\ \epsilon_{\mathbf{b}} \end{pmatrix}. \tag{19}$$

Noting that the top right block of the left hand matrix is zero, $\delta_{\mathbf{a}}$ can be determined from the top half of the above equations, which is

$$(\mathbf{U}^* - \mathbf{W}\ \mathbf{V}^{*-1}\ \mathbf{W}^T)\ \delta_{\mathbf{a}} = \epsilon_{\mathbf{a}} - \mathbf{W}\ \mathbf{V}^{*-1}\ \epsilon_{\mathbf{b}}. \tag{20}$$

Having solved for $\delta_{\mathbf{a}}$, $\delta_{\mathbf{b}}$ can then be computed by backsubstitution into the bottom half of Eq. (19), which yields

$$\mathbf{V}^*\ \delta_{\mathbf{b}} = \epsilon_{\mathbf{b}} - \mathbf{W}^T\ \delta_{\mathbf{a}}. \tag{21}$$

Substituting $\mathbf{U}^*$, $\mathbf{W}$ and $\mathbf{V}^*$ from Eq. (16), the matrix within parentheses in the left hand side of Eq. (20) equals

$$\begin{pmatrix} \mathbf{U}_1^* & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2^* & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{U}_3^* \end{pmatrix} - \begin{pmatrix} \mathbf{W}_{11} & \mathbf{W}_{21} & \mathbf{W}_{31} & \mathbf{W}_{41} \\ \mathbf{W}_{12} & \mathbf{W}_{22} & \mathbf{W}_{32} & \mathbf{W}_{42} \\ \mathbf{W}_{13} & \mathbf{W}_{23} & \mathbf{W}_{33} & \mathbf{W}_{43} \end{pmatrix} \begin{pmatrix} \mathbf{V}_1^{*-1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_2^{*-1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{V}_3^{*-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{V}_4^{*-1} \end{pmatrix} \begin{pmatrix} \mathbf{W}_{11}{}^T & \mathbf{W}_{12}^T & \mathbf{W}_{13}^T \\ \mathbf{W}_{21}{}^T & \mathbf{W}_{22}^T & \mathbf{W}_{23}^T \\ \mathbf{W}_{31}{}^T & \mathbf{W}_{32}^T & \mathbf{W}_{33}^T \\ \mathbf{W}_{41}{}^T & \mathbf{W}_{42}^T & \mathbf{W}_{43}^T \end{pmatrix}. \quad (22)$$

Letting $\mathbf{Y}_{ij} = \mathbf{W}_{ij}\mathbf{V}_i^{*-1}$, (22) becomes

$$\begin{pmatrix} \mathbf{U}_1^* - \sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i1}{}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i2}{}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i3}{}^T \\ -\sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i1}{}^T & \mathbf{U}_2^* - \sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i2}{}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i3}{}^T \\ -\sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i1}{}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i2}{}^T & \mathbf{U}_3^* - \sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i3}{}^T \end{pmatrix}. \quad (23)$$

Also, the right hand side of Eq. (20) equals

$$\epsilon_{\mathbf{a}} - \begin{pmatrix} \sum_{i=1}^{4} \mathbf{Y}_{i1}\ \epsilon_{\mathbf{b}_i} \\ \sum_{i=1}^{4} \mathbf{Y}_{i2}\ \epsilon_{\mathbf{b}_i} \\ \sum_{i=1}^{4} \mathbf{Y}_{i3}\ \epsilon_{\mathbf{b}_i} \end{pmatrix}. \quad (24)$$

Combining (23) and (24), $\delta_{\mathbf{a}}$ can be computed by solving the system

$$\begin{pmatrix} \mathbf{U}_1^* - \sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i1}{}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i2}{}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i1}\mathbf{W}_{i3}{}^T \\ -\sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i1}{}^T & \mathbf{U}_2^* - \sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i2}{}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i2}\mathbf{W}_{i3}{}^T \\ -\sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i1}{}^T & -\sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i2}{}^T & \mathbf{U}_3^* - \sum_{i=1}^{4} \mathbf{Y}_{i3}\mathbf{W}_{i3}{}^T \end{pmatrix} \begin{pmatrix} \delta_{\mathbf{a}_1} \\ \delta_{\mathbf{a}_2} \\ \delta_{\mathbf{a}_3} \end{pmatrix} = \begin{pmatrix} \epsilon_{\mathbf{a}_1} - \sum_{i=1}^{4} \mathbf{Y}_{i1}\ \epsilon_{\mathbf{b}_i} \\ \epsilon_{\mathbf{a}_2} - \sum_{i=1}^{4} \mathbf{Y}_{i2}\ \epsilon_{\mathbf{b}_i} \\ \epsilon_{\mathbf{a}_3} - \sum_{i=1}^{4} \mathbf{Y}_{i3}\ \epsilon_{\mathbf{b}_i} \end{pmatrix}. \quad (25)$$

Left multiplication of Eq. (21) by $\mathbf{V}^*$ yields $\delta_{\mathbf{b}}$ as

$$\delta_{\mathbf{b}} = \begin{pmatrix} \mathbf{V}_1^{*-1}\ (\epsilon_{\mathbf{b}_1} - \sum_{j=1}^{3} \mathbf{W}_{1j}{}^T\ \delta_{\mathbf{a}_j}) \\ \mathbf{V}_2^{*-1}\ (\epsilon_{\mathbf{b}_2} - \sum_{j=1}^{3} \mathbf{W}_{2j}{}^T\ \delta_{\mathbf{a}_j}) \\ \mathbf{V}_3^{*-1}\ (\epsilon_{\mathbf{b}_3} - \sum_{j=1}^{3} \mathbf{W}_{3j}{}^T\ \delta_{\mathbf{a}_j}) \\ \mathbf{V}_4^{*-1}\ (\epsilon_{\mathbf{b}_4} - \sum_{j=1}^{3} \mathbf{W}_{4j}{}^T\ \delta_{\mathbf{a}_j}) \end{pmatrix}. \quad (26)$$

At this point it should be evident that the solution of the normal equations that was illustrated above can be directly generalized to arbitrary n and m. Note that if a point k does not appear in an image l then $\mathbf{A}_{kl} = \mathbf{0}$ and $\mathbf{B}_{kl} = \mathbf{0}$. Index i in the summations appearing in the definitions of $\mathbf{U}_j$ and $\epsilon_{\mathbf{a}_j}$ 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 projects on. Fig. 2 summarizes the general procedure for solving the sparse normal equations involved in the LM algorithm[3]. This procedure can be embedded into the LM algorithm of section 2 at the point indicated by the rectangular box in Fig. 1, leading to a sparse bundle adjustment algorithm.

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 j-th 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 in $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ respectively.

Compute the following auxiliary variables:

$$\mathbf{U}_j := \sum_i \mathbf{A}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{A}_{ij} \quad \mathbf{V}_i := \sum_j \mathbf{B}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{B}_{ij} \quad \mathbf{W}_{ij} := \mathbf{A}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \mathbf{B}_{ij}$$

$$\epsilon_{\mathbf{a}_j} := \sum_i \mathbf{A}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \epsilon_{ij} \quad \epsilon_{\mathbf{b}_i} := \sum_j \mathbf{B}_{ij}{}^T \mathbf{\Sigma}_{\mathbf{x}_{ij}}^{-1} \epsilon_{ij}$$

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

Figure 2: Algorithm for solving the sparse normal equations arising in generic bundle adjustment; see text for details.

[3]The original presentation in [9] contains a few typographic errors, which have been corrected in the description included here.

4 Implementation Details

This section provides details regarding the practical implementation of the sparse bundle adjustment algorithm sketched in sections 2 and 3. 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 4.1, 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 4.2, are designed for the less knowledgeable user who is willing to trade some potential loss in performance for increased ease of use.

4.1 Expert Drivers

Let us begin by considering the matrices $\mathbf{A}$ and $\mathbf{B}$ consisting respectively of the blocks $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ defined in the algorithm of Fig. 2. Note that if a point i does not appear in image j, then $\mathbf{A}_{ij} = \mathbf{B}_{ij} = \mathbf{0}$, which implies that $\mathbf{A}$ and $\mathbf{B}$ are sparse. For instance, referring to the sample problem with $n = 4$ and $m = 3$ outlined in section 3, 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 (27); $\mathbf{B}$ has a similar structure.

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} \\ \mathbf{0} & \mathbf{A}_{42} & \mathbf{A}_{43} \end{pmatrix}, \quad \mathcal{I} = \begin{pmatrix} 0 & 1 & -1 \\ -1 & 2 & 3 \\ -1 & 4 & 5 \\ -1 & 6 & 7 \end{pmatrix} \tag{27}$$

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 by placing its non-zero blocks $\mathbf{A}_{ij}$ one after the other in a contiguous memory buffer and a matrix $\mathcal{I}$ is set up 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 (27). Indices stored in $\mathcal{I}$ conform to the C convention and start from 0. 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 special techniques can be used for storing it.

In order 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 non-zero matrix elements along with some additional information for knowing 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 [2] 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 non-zero 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 non-zero array elements. As an example, the CRS vectors for the 4×3 matrix $\mathcal{I}$ with 8 non-zero elements defined above, are shown below. Again, array indices conform to the C convention and start from 0:

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

Thus, the `sba` routines store $\mathbf{A}$ by keeping its non-zero $\mathbf{A}_{ij}$ 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 the $\mathbf{B}_{ij}$ and the corresponding contiguous memory block indices. Note also that blocks $\mathbf{W}_{ij}$ and $\mathbf{Y}_{ij}$ defined in the algorithm of Fig. 2 are zero if either of $\mathbf{A}_{ij}$, $\mathbf{B}_{ij}$ 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 storage strategy employed for matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{W}$ and $\mathbf{Y}$ makes efficient use of the processor's cache since the elements of non-zero blocks are kept in consecutive memory locations which are very likely to fit in a single cache block. 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 Eqs. (6) and (7) 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 $\mathbf{X}$, $\hat{\mathbf{X}}$. File `sba_crsm.c` in the `sba` package provides several routines for manipulating CRS sparse matrices. The CRS format is represented in C/C++ by 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;  /* locations in val that start a row. size: nr+1 and rowptr[nr]=nnz. */
};
```

Sparse BA is implemented in `sba` by the expert function `sba_motstr_levmar_x()`. The prototype of `sba_motstr_levmar_x()` from `sba.h` is

```
int sba_motstr_levmar_x(const int n, const int m, const int mcon, char *vmask, double *p,
   const int cnp, const int pnp, double *x, const 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[3], double info[10]);

In case of successful termination, the function returns the number of iterations required for the minimization (≥ 0), otherwise -1. The current implementation assumes that $\mathbf{\Sigma_X}$, the covariance matrix of the measurement vector, is equal to the identity matrix. The linear system of Eq. (25) is solved with the aid of LU factorization, implemented using appropriate LAPACK [1] routines. For experimenting with other approaches, sba also includes implementations of LAPACK-based linear system solvers employing QR and SVD decomposition. 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. When the number of cameras involved in bundle adjustment is very large, solving Eq. (25) with any of the LAPACK-based solvers will be less efficient compared to employing a solver based on the conjugate gradients (CG) method. Thus, sba incorporates an iterative linear systems solver that is based on conjugate gradients with jacobi or SSOR preconditioning [23, 2]. `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 $[\mathbf{I} \mid \mathbf{0}]$. (I)

`vmask` : Point visibility mask: `vmask[(i-1)*m+j-1]=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 0, thus the `-1`'s in the expression `(i-1)*m+j-1`. The size of `vmask` is `nxm`. (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 6 parameters (3 rotational + 3 translational). If quaternions are used for the rotations, the number of parameters increases to 7 (i.e. 4 + 3). A fully projective camera can be parameterized with 11 or, including the scale factor, with 12 parameters. (I)

`pnp` : The number of parameters defining a single 3D point: e.g. 3 for Euclidean points, 4 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 maximum size is `n*m*mnp`. (I)

`mnp` : The number of parameters defining an image point (typically 2). (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 (5) 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 returned as $(\mathbf{A}_{11}, \ldots, \mathbf{A}_{1m}, \ldots, \mathbf{A}_{n1}, \ldots, \mathbf{A}_{nm}, \mathbf{B}_{11}, \ldots, \mathbf{B}_{1m}, \ldots, \mathbf{B}_{n1}, \ldots, \mathbf{B}_{nm})$, 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 $\mathbf{A}_{ij}$ (resp. $\mathbf{B}_{ij}$) blocks is made up of `mnp*cnp` (resp. `mnp*pnp`) elements and is stored in a row-major order, occupying a distinct storage block in `jac`. The $\mathbf{A}_{ij}$ are stored in a segment at the beginning of the `jac` array whereas the $\mathbf{B}_{ij}$ are placed in a second segment that starts right after the first one. Both segments consist of `idxij->nnz` nonzero blocks. The exact offset of each $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ into its corresponding segment 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. If the user specifies a `NULL` value for `fjac`, the Jacobian is approximated using finite differences on data provided by successive invocations of `func` (see function `sba_fdjac_x()` for the corresponding code). However, the approximation of the Jacobian requires `m*cnp+n*pnp+1` calls to `func`, each of which estimates the image projections of all points in all cameras. Clearly, while this approach might be acceptable for initial testing and debugging, it is not efficient computationally. Therefore, when execution time is a concern, the jacobian should be computed analytically by a user-supplied function. (I)

`adata` : Pointer to possibly additional data, passed uninterpreted to `func`, `fjac`. It is intended to help avoid direct use of globals 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 uninterpreted to each call of the user-supplied routines. This argument can be set to NULL if not needed. (I)

`itmax` : Maximum number of Levenberg-Marquardt iterations (k_{max} in the algorithm of Fig. 1). (I)

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

`opts` : Minimization options $\tau, \varepsilon_1, \varepsilon_2$ for the Levenberg-Marquardt algorithm (see Fig. 1). Respectively the scale factor for the initial damping term and the stopping tolerance thresholds. (I)

`info` : Information regarding the outcome of the minimization. It can be set to `NULL` if not needed. (O)

`info[0]` : $||\epsilon_{\mathbf{p}_0}||^2$, i.e. 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 non-zeros in `vmask`) corresponds to the initial mean squared reprojection error.

`info[1-4]` : $(||\epsilon_{\mathbf{p}}||^2, ||\mathbf{J}^T\epsilon_{\mathbf{p}}||_\infty, ||\delta_{\mathbf{p}}||^2, \mu / \max_k(\left[\mathbf{J}^T\mathbf{J}\right]_{kk}))$, 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` : The matrix of the augmented normal equations is singular, minimization should be restarted from the current solution with an increased damping term.

`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 (see again the innermost loop of the algorithm in Fig. 1).

Remark that all information pertaining to the BA is selectable by the user of `sba_motstr_levmar_x()`: 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, i.e. the problem of estimating the camera matrix from the 2D image projections of a set of 3D points that are assumed fixed and precisely known [15, 10]. The prototype of `sba_mot_levmar_x()` is the following:

```
int sba_mot_levmar_x(const int n, const int m, const int mcon, char *vmask, double *p,
    const int cnp, double *x, const 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[3], double info[10]);
```

Function `sba_mot_levmar_x()` implements the algorithm resulting from that in Fig. 2 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 Eq. (17) are simplified to a set of linear systems $\mathbf{U}_j^* \, \delta_{\mathbf{a}_j} = \epsilon_{\mathbf{a}_j}$, which can be solved with either any of the LAPACK-based LU, QR or SVD solvers or the CG iterative solver. Furthermore, taking into account that the matrices $\mathbf{U}_j^*$ are symmetric, the previous systems can also be solved with the aid of Cholesky or Bunch-Kaufman factorization; such solvers are also included in `sba`. 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 the inverse of `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. Function `sba_str_levmar_x()` implements the algorithm resulting from that in Fig. 2 after setting $\mathbf{A}_{ij} = \mathbf{U}_j = \mathbf{W}_{ij} = \mathbf{Y}_{ij} = \mathbf{0}$ and its prototype is as follows:

```
int sba_str_levmar_x(const int n, const int m, char *vmask, double *p,
    const int pnp, double *x, const 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[3], double info[10]);
```

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

4.2 Simple Drivers

To users that are not willing to spent much time understanding its inner workings, `sba` offers three simple drivers, namely `sba_motstr_levmar()`, `sba_mot_levmar()` and `sba_str_levmar()`, that are wrappers around the expert drivers `sba_motstr_levmar_x()`, `sba_mot_levmar_x()` and `sba_str_levmar_x()` respectively. They differ from the later 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 measurement vector and its sparse jacobian are then estimated by repeatedly invoking `proj` and `projac` for all points and cameras. Thus, at the cost of the potentially extra overhead induced by the higher function call count, the simple drivers free the user from worrying about how is the measurement vector and its sparse jacobian laid out in memory internally.

The prototype of `sba_motstr_levmar()` is

```
int sba_motstr_levmar(const int n, const int m, const int mcon, char *vmask, double *p,
    const int cnp, const int pnp, double *x, const 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[3], double info[10]);
```

that of `sba_mot_levmar()`

```
int sba_mot_levmar(const int n, const int m, const int mcon, char *vmask, double *p,
    const int cnp, double *x, const 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[3], double info[10]);
```

and that of `sba_str_levmar()`

```
int sba_str_levmar(const int n, const int m, char *vmask, double *p,
    const int pnp, double *x, const 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[3], double info[10]);
```

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 $\frac{\partial \mathbf{Q}(\mathbf{a}_j,\mathbf{b}_i)}{\partial \mathbf{b}_i}$, i.e. 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 knowledge 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()`.

5 A Sample Use Case

An example of using the developed BA routines is presented in this section. The example concerns the use of `sba` for Euclidean bundle adjustment. 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 camera motions are available. The remainder of this section describes the application of `sba` for refining those motion and structure estimates.

The employed world coordinate frame is taken to be aligned with the initial camera 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 $\mathbf{R} = (\cos(\frac{\theta}{2}),\ u_1 \sin(\frac{\theta}{2}),\ u_2 \sin(\frac{\theta}{2}),\ u_3 \sin(\frac{\theta}{2}))$ [26]. A 3D translation is defined by a vector $\mathbf{t}$. A 3D point is represented by its Euclidean coordinate vector $\mathbf{M}$. Thus, the parameters of each camera j and point i are $\mathbf{a}_j = (\mathbf{R}_j, \mathbf{t}_j{}^T)^T$ and $\mathbf{b}_i = \mathbf{M}_i$, respectively. With the previous definitions, the predicted projection of point i on image j is

$$\mathbf{Q}(\mathbf{a}_j, \mathbf{b}_i) = \mathbf{K}\ (\mathbf{R}_j\ \mathbf{N}_i\ \mathbf{R}_j^{-1}\ +\ \mathbf{t}_j), \tag{28}$$

where $\mathbf{K}$ is the 3×3 intrinsic camera calibration matrix and $\mathbf{N}_i = (0,\ \mathbf{M}_i{}^T)$ is the vector quaternion corresponding to the 3D point $\mathbf{M}_i$. The expression $\mathbf{R}_j\ \mathbf{N}_i\ \mathbf{R}_j^{-1}$ corresponds to point $\mathbf{M}_i$ rotated by an angle θ_j about unit vector $\mathbf{u}_j$, as specified by the quaternion $\mathbf{R}_j$. Source file `eucsbademo.c` accompanying the `sba` package implements routines for evaluating the estimated measurement vector and its jacobian with respect to all camera and 3D point parameters. These can 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 Eq. (28), whose code was generated automatically using MAPLE's symbolic differentiation facilities[5]. 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 $\mathbf{K}\ [\mathbf{I}_{3\times3} \mid \mathbf{0}]$.

File `eucsbademo.c` also provides a demo program illustrating the use of both the expert (i.e. `sba_motstr_levmar_x()` and `sba_mot_levmar_x()`) and the simple

[4] `XXX` stands for `motstr` or `mot`.

[5] 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 [8].

(i.e. `sba_motstr_levmar()` and `sba_mot_levmar()`) sparse BA driver routines for Euclidean BA. In all cases, the initial estimates of the camera parameters are read from a text file that has a separate line for every camera, each line containing 7 motion parameters (4 for rotation and 3 for translation). The initial point structure estimates and their image projections are also read from a text file. Each line of this file corresponds to a single 3D point and its image projections and has the format `X Y Z nframes frame0 x0 y0 frame1 x1 y1 ...`, where `X Y Z` are the points' Euclidean 3D coordinates, `nframes` is the total number of frames (i.e. images) in which the points' projections have been identified and each of the `nframes` subsequent triplets of the form `frame x y` specifies that the 3D point in question projects to pixel `x y` in frame `frame`. For instance, the line

```
100.0 200.0 300.0 3 2 270.0 114.1 4 234.2 321.7 5 173.6 425.8
```

refers to 3D point (100.0, 200.0, 300.0) projecting to the 3 points (270.0, 114.1), (234.2, 321.7) and (173.6, 425.8) in images 2, 4 and 5 respectively. Notice that this format allows the image projections of a 3D point to be specified for any subset of the available images. Both camera and 3D point indices start from 0. One particular BA problem that is included as a test case involves 54 cameras and 5207 3D points that give rise to 24609 image projections. The corresponding minimization problem depends upon 15999 variables and was solved in about 13 sec on a Intel P4@1.8 GHz running Linux. Without a sparse implementation of BA, a problem of this size would simply be intractable. More details regarding this application of `sba` can be found by studying the relevant supplied source code and the comments annotating it.

6 Conclusions

This paper has presented the mathematical theory behind an 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 successfully demonstrated its ability to deal efficiently with very large BA problems. The package has been made freely available in the hope to be useful to the computer vision community. To the best of our knowledge, `sba` is the first and currently the only such software package to be placed in the public domain.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, 3rd edition, 1999.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates*

for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia, PA, 2nd edition, 1994. HTML version at `http://www.netlib.org/linalg/html_templates/report.html`.

[3] P. Beardsley, P.H.S. Torr, and A. Zisserman. 3D Model Acquisition From Extended Image Sequences. In *Proc. of ECCV'96*, pages 683–695, 1996.

[4] J.E. Dennis. Nonlinear Least-Squares. In D. Jacobs, editor, *State of the Art in Numerical Analysis*, pages 269–312. Academic Press, 1977.

[5] J.E. Dennis, D.M. Gay, and R.E. Welsch. An Adaptive Nonlinear Least-Squares Algorithm. *ACM Trans. on Math. Software*, 7(3):348–368, Sep. 1981.

[6] A.W. Fitzgibbon and A. Zisserman. Automatic Camera Recovery for Closed or Open Image Sequences. In *Proceedings of ECCV'98*, pages 311–326, 1998.

[7] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 3rd edition, 1996.

[8] A. Griewank, D. Juedes, and J. Utke. ADOL–C, A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, 1996.

[9] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, 1st edition, 2000.

[10] R.I. Hartley. Euclidean Reconstruction from Uncalibrated Views. In J. Mundy and A. Zisserman, editors, *Applications of Invariance in Computer Vision*, volume 825 of Lecture Notes in Computer Science, pages 237–256. Springer-Verlag, 1993.

[11] K.L. Hiebert. An Evaluation of Mathematical Software that Solves Nonlinear Least Squares Problems. *ACM Trans. on Math. Software*, 7(1):1–16, Mar. 1981.

[12] M. Lampton. Damping-Undamping Strategies for the Levenberg-Marquardt Nonlinear Least-Squares Method. *Computers in Physics Journal*, 11(1):110–115, Jan./Feb. 1997.

[13] K. Levenberg. A Method for the Solution of Certain Non-linear Problems in Least Squares. *Quarterly of Applied Mathematics*, 2(2):164–168, Jul. 1944.

[14] M.I.A. Lourakis and A.A. Argyros. Efficient 3D Camera Matchmoving Using Markerless, Segmentation-Free Plane Tracking. Technical Report 324, Institute of Computer Science - FORTH, Heraklion, Greece, Aug. 2003. Available at `ftp://ftp.ics.forth.gr/tech-reports/2003`.

[15] C.-P. Lu, G.D. Hager, and E. Mjolsness. Fast and Globally Convergent Pose Estimation from Video Images. *IEEE Trans. on PAMI*, 22(6):610–622, Jun. 2000.

[16] K. Madsen, H.B. Nielsen, and O. Tingleff. Methods for Non-Linear Least Squares Problems. Technical University of Denmark, 2004. Lecture notes, available at `http://www.imm.dtu.dk/courses/02611/nllsq.pdf`.

[17] D.W. Marquardt. An Algorithm for the Least-Squares Estimation of Nonlinear Parameters. *SIAM Journal of Applied Mathematics*, 11(2):431–441, Jun. 1963.

[18] H.D. Mittelmann. The Least Squares Problem. [web page] `http://plato.asu.edu/topics/problems/nlolsq.html`, Jul. 2004. [Accessed on 4 Aug. 2004.].

[19] J.J. Moré, B.S. Garbow, and K.E. Hillstrom. User guide for MINPACK-1. Technical Report ANL-80-74, Argonne National Laboratory, Aug. 1980.

[20] H.B. Nielsen. Damping Parameter in Marquardt's Method. Technical Report IMM-REP-1999-05, Technical University of Denmark, 1999. Available at `http://www.imm.dtu.dk/~hbn`.

[21] M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, and R. Koch. Visual Modeling With a Hand-Held Camera. *IJCV*, 59(3):207–232, Sep./Oct. 2004.

[22] W.H. Press, S.A. Teukolsky, A.W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, 1992.

[23] J.R. Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical Report CMU-CS-94-125, School of Computer Science, Carnegie Mellon University, 1994.

[24] C.C. Slama. *Manual of Photogrammetry.* American Society of Photogrammetry, Falls Church, VA, fourth edition, 1980.

[25] B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon. Bundle Adjustment – A Modern Synthesis. In *Proceedings of the International Workshop on Vision Algorithms: Theory and Practice*, pages 298–372, 1999.

[26] L. Vicci. Quaternions and Rotations in 3-Space: The Algebra and its Geometric Interpretation. Technical Report TR01-014, Dept. of Computer Science, University of North Carolina at Chapel Hill, 2001.

[27] Z. Zhang and Y. Shan. Incremental Motion Estimation through Local Bundle Adjustment. Technical Report MSR-TR-01-54, Microsoft Research, May 2001.