Efficient Computation of Posterior Covariance in Bundle Adjustment in DBAT for Projects with Large Number of Object Points

Niclas Börlin, Arnadi Murtiyoso, Pierre Grussenmeyer

EFFICIENT COMPUTATION OF POSTERIOR COVARIANCE IN BUNDLE ADJUSTMENT IN DBAT FOR PROJECTS WITH LARGE NUMBER OF OBJECT POINTS

N. Börlin[1,*], A. Murtiyoso[2], P. Grussenmeyer[2]

[1] Department of Computing Science, Umeå University, Sweden — niclas.borlin@cs.umu.se
[2] Photogrammetry and Geomatics Group, ICube Laboratory UMR 7357, INSA Strasbourg, France — (arnadi.murtiyoso, pierre.grussenmeyer)@insa-strasbourg.fr

Commission II, WG II/7

KEY WORDS: Bundle adjustment, Quality control, Posterior covariance, Software, Photogrammetry

ABSTRACT:

One of the major quality control parameters in bundle adjustment are the posterior estimates of the covariance of the estimated parameters. Posterior covariance computations have been part of the open source Damped Bundle Adjustment Toolbox in Matlab (DBAT) since its first public release. However, for large projects, the computation of especially the posterior covariances of object points have been time consuming.
The non-zero structure of the normal matrix depends on the ordering of the parameters to be estimated. For some algorithms, the ordering of the parameters highly affect the computational effort needed to compute the results. If the parameters are ordered to have the object points first, the non-zero structure of the normal matrix forms an arrowhead.
In this paper, the legacy DBAT posterior computation algorithm was compared to three other algorithms: The Classic algorithm based on the reduced normal equation, the Sparse Inverse algorithm by Takahashi, and the novel Inverse Cholesky algorithm. The Inverse Cholesky algorithm computes the explicit inverse of the Cholesky factor of the normal matrix in arrowhead ordering.
The algorithms were applied to normal matrices of ten data sets of different types and sizes. The project sizes ranged from 21 images and 100 object points to over 900 images and 400,000 object points. Both self-calibration and non-self-calibration cases were investigated. The results suggest that the Inverse Cholesky algorithm is the fastest for projects up to about 300 images. For larger projects, the Classic algorithm is faster. Compared to the legacy DBAT implementation, the Inverse Cholesky algorithm provides a performance increase by one to two orders of magnitude. The largest data set was processed in about three minutes on a five year old workstation.
The legacy and Inverse Cholesky algorithms were implemented in Matlab. The Classic and Sparse Inverse algorithms included code written in C. For a general toolbox as DBAT, a pure Matlab implementation is advantageous, as it removes any dependencies on, e.g., compilers. However, for a specific lab with mostly large projects, compiling and using the classic algorithm will most likely give the best performance. Nevertheless, the Inverse Cholesky algorithm is a significant addition to DBAT as it enables a relatively rapid computation of more statistical metrics, further reinforcing its application for reprocessing bundle adjustment results of black-box solutions.

1. INTRODUCTION

1.1 Background

One of the major quality control parameters in bundle adjustment are the posterior estimates of the covariance of the estimated parameters. In photogrammetric projects oriented towards robust measuring purposes, this is not only essential but very important for quality control. Indeed, the covariance values indicate the quality of the bundle adjustment result and thus its feasibility for use as topographic and surveying products. A common practice would be to compare these posterior covariance values with the project requirements, which often times depends on the required map scale. A photogrammetric mission may be considered acceptable if and only if its quality parameters, including the covariance values, are within the project's tolerance.

A more in-depth analysis to the geometrical qualities of a photogrammetric project is also sometimes necessary, for example when errors or deviation from prior values are detected. This becomes more and more important with the increasing use of UAV (Unmanned Aerial Vehicle) platforms, colloquially known as drones. Contrary to classical metric photogrammetric survey, the use of consumer-grade cameras in drone imagery is less reliable and therefore requires more attention as far as the quality is concerned. This in-depth analysis may be performed by various means, including by computing the correlation rate between the different calculated parameters; a metric which is also derived from the covariance matrix.

Since geometric quality is of the utmost importance in these spatial applications, the computation of covariance is very interesting for quality control purposes. However, for large projects, the computation of especially the posterior covariances of object points (OP) can be a time consuming process.

The Damped Bundle Adjustment Toolbox (DBAT) for Matlab was conceived as an open source toolbox to perform bundle adjustment computations (Börlin and Grussenmeyer, 2013). It has been used for several applications, most notably to reprocess the bundle adjustment results of commercial solutions (Murtiyoso et al., 2018). The main highlight of DBAT has always been its modularity (Börlin et al., 2019) and openness, with the possibility to access bundle adjustment metrics contrary to the black-box nature of several commercial software.

*Corresponding author

1.2 Related work

Various schemes that exploit the sparsity of the normal matrix to speed up the bundle adjustment computations were presented already in the early days of Bundle Adjustment (Brown, 1968, 1976) and are now part of standard photogrammetric textbooks (e.g. Kraus, 2007; Wolf et al., 2013; Luhmann et al., 2014; Förstner and Wrobel, 2016).

In most cases, the presentations are focused on speeding up the computations during the bundle adjustment iterations, where the linearised equation has a single right-hand side. In contrast, the computation of the posterior covariance has a large number of right-hand sides, and said algorithm cannot be immediately applied. Literature that discusses algorithms for posterior covariance computations include Triggs et al. (2000); Wolf et al. (2013); Förstner and Wrobel (2016); Ila et al. (2017). However, the details of how the sparsity can be exploited are rarely given.

The method by Takahashi et al. (1980) can be used to compute subsets of the inverse of a sparse matrix. The sparse inverse (SI) method was developed for electrical networks, but can be applied to any sparse matrix problem.

Recently, Kallin (2019) presented an algorithm that computed the explicit inverse of the Cholesky factorisation of the normal matrix using a combination of sparse and dense computations. Depending on the density of a matrix, i.e., the fraction of non-zeros, dense computations can have performance advantage over sparse computations due to their more efficient use of the memory hierarchy Duff et al. (2002); Goto and Van De Geijn (2008). Furthermore, the dense algorithms are easier to parallellise due to their predictable memory access patterns. The inverse Cholesky algorithm (called VDSIBlock in Kallin (2019)) was shown to outperform the SI algorithm on small-to-medium-sized data sets.

1.3 Aim

The aim of this paper is to extend the investigation of the Inverse Cholesky algorithm to larger data sets and to compare the results with the current DBAT implementation and classic photogrammetric algorithms.

2. THE STRUCTURE OF THE NORMAL MATRIX

2.1 Non-zero pattern

The normal matrix has a characteristic pattern of non-zero elements. If the parameters are ordered with the external orientation (EO) parameters first, the normal matrix may be blocked as

$$N = \begin{pmatrix} N_{ee} & N_{eo} \\ N_{oe} & N_{oo} \end{pmatrix}. \tag{1}$$

The internal block structure is visualised in Figure 1. The N_{ee} and N_{oo} blocks are block diagonal with 6-by-6 and 3-by-3 blocks, respectively. The non-zero coefficients of each block relate only to the EO or OP parameters of a single camera or point, respectively. In contrast, the N_{eo} block contain 3-by-6 blocks that connect images to object points.

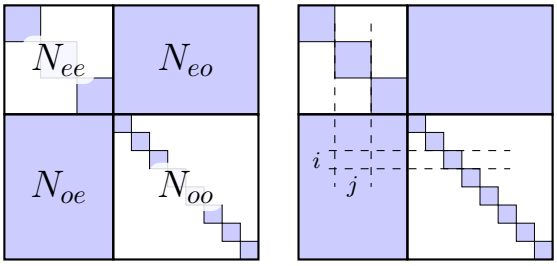

Figure 1: Non-zero structure of the normal matrix with the EO parameters first (left). The N_{ee} block has 6-by-6 blocks that each correspond to the EO parameters of one camera. Similarly for N_{oo} and object points, but with 3-by-3 blocks. The N_{oe} block consists of 3-by-6 blocks that connect an image with an object point. Non-zeros can only appear at block row i, block column j of N_{eo} if object point i was measured in image j (right).

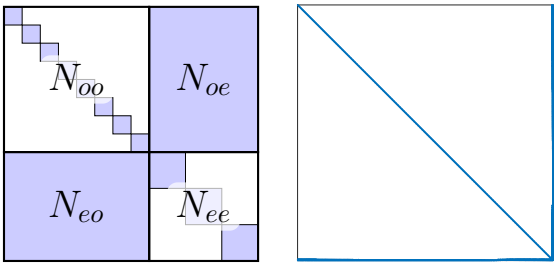

Figure 2: If the object points are ordered first, the normal matrix forms an arrowhead matrix, where the N_{oo} block forms the shaft of the arrow, and the remaining blocks form the tip at the lower and right borders of the matrix (left). The arrowhead shape is more clear for real projects, where the number of OP make the N_{oo} block dominate the matrix. The right figure shows the normal matrix for the ROMA data set (see Section 4.1) with 60 images and about 25000 object points.

2.2 Arrowhead permutation

If the ordering of the unknown parameters is changed to have the object points first, the blocks of the normal matrix are swapped to

$$N = \begin{pmatrix} N_{oo} & N_{oe} \\ N_{eo} & N_{ee} \end{pmatrix}. \tag{2}$$

With this ordering, the non-zero structure of the matrix has the shape of an arrowhead, where the large N_{oo} block form the shaft of the arrow, and the remaining block form the ”tip” of the arrow at the lower and right borders (Figure 2).

2.3 Self-calibration

For a self-calibration project, the normal matrix also include blocks for the internal orientation (IO) parameters. With the IO parameters first, the blocking becomes

$$N = \begin{pmatrix} N_{ii} & N_{ie} & N_{io} \\ N_{ei} & N_{ee} & N_{eo} \\ N_{oi} & N_{oe} & N_{oo} \end{pmatrix}, \tag{3}$$

where the new blocks have the width corresponding to the number of IO parameters and are typically fully dense. The corresponding arrowhead ordering is

$$N = \begin{pmatrix} N_{oo} & N_{oe} & N_{oi} \\ N_{eo} & N_{ee} & N_{ei} \\ N_{io} & N_{ie} & N_{ii} \end{pmatrix}. \tag{4}$$

See Figure 3.

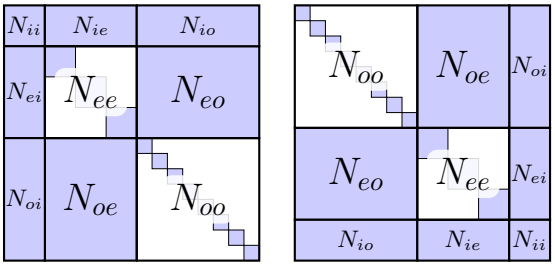

Figure 3: A self-calibration normal matrix has an added border of dense blocks. The blocks have a thickness equal to the number of estimated IO parameters. In the standard ordering (left), the IO blocks form the left and upper border of the matrix. In the arrowhead ordering, the IO blocks instead form the lower and right border.

Algorithm 1 The classic algorithm for computing posterior covariance for object points. The sparse and dense steps are computed in Matlab using sparse and dense linear algebra routines, respectively. The C code step is computed by compiled C-code called from Matlab ("mex" code).

Require: N has standard ordering

1: $A \leftarrow N_{ee}$. ▷ Extract blocks
2: $B \leftarrow N_{eo}$.
3: $C \leftarrow N_{oo}$.
4: $E \leftarrow D^{-1}$ ▷ Sparse
5: $F \leftarrow BEB^T$ ▷ Sparse
6: $C_{ee} \leftarrow F^{-1}$ ▷ Dense
7: $G \leftarrow BE$ ▷ Sparse
8: $H \leftarrow G^T C_{ee} G$ ▷ C code, 3-by-3 diagonal blocks only
9: $C_{oo} \leftarrow G + H$ ▷ Sparse

3. ALGORITHMS

3.1 Classic

The classic algorithm, as described in Wolf et al. (2013, Ch. 17-12) uses the normal matrix N in standard ordering (eq. (1)):

$$N = \begin{pmatrix} N_{ee} & N_{eo} \\ N_{oe} & N_{oo} \end{pmatrix} = \begin{pmatrix} A & B \\ B^T & D \end{pmatrix}, \tag{5}$$

where m and n are the number of images and object points, respectively.

Given the same partitioning of the posterior covariance

$$N^{-1} = C = \begin{pmatrix} C_{ee} & C_{eo} \\ C_{oe} & C_{oo} \end{pmatrix}, \tag{6}$$

the matrix C_{oo} contain the posterior covariance for the object points. Mathematically, the matrix C_{oo} can be computed as follows (cf. (Wolf et al., 2013, eq. (17-34–36))):

$$C_{ee} = (A - BD^{-1}B^T)^{-1}, \tag{7}$$
$$C_{oo} = D^{-1} + D^{-1}B^T C_{ee} B D^{-1}. \tag{8}$$

The actual computation is given in Algorithm 1.

3.2 Inverse Cholesky

The Inverse Cholesky algorithm uses the arrowhead ordering (eq. (2))

$$N = \begin{pmatrix} N_{oo} & N_{eo}^T \\ N_{eo} & N_{ee} \end{pmatrix}, \tag{9}$$

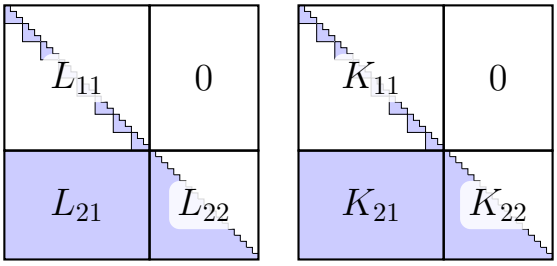

Figure 4: The non-zero patterns of the Cholesky factor L (left) forms half an arrowhead. The L_{11} block is block diagonal with 3-by-3 lower triangular blocks. Due to the arrowhead shape of the normal matrix, the fill-in during the Cholesky factorisation is restricted to the lower triangular L_{22} block. The L_{21} block does not experience any fill-in at all. The non-zero pattern of the Cholesky inverse $K = L^{-1}$ (right) is similar to that of L, except that the K_{21} and K_{22} block may experience fill-in. The K_{22} block will typically be close to half full.

The normal matrix is Cholesky factorised into $LL^T = N$ with

$$L = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}. \tag{10}$$

The non-zero structure of L forms half an arrowhead (Figure 4, left). The L_{11} block will be 3-by-3 block diagonal with lower triangular blocks. Due to the arrowhead structure of the normal matrix, there will be no fill-in in the L_{21} block during the Cholesky factorisation. Instead, the fill-in is restricted to the lower triangular L_{22} block.

The inverse K of the Cholesky factor is

$$K = L^{-1} = \begin{pmatrix} K_{11} & 0 \\ K_{21} & K_{22} \end{pmatrix}. \tag{11}$$

This leads to the following equations:

$$K_{11} = L_{11}^{-1}, \tag{12}$$
$$K_{21} = -L_{22}^{-1} L_{21} K_{11}, \tag{13}$$
$$K_{22} = L_{22}^{-1}. \tag{14}$$

The K_{11} block can be computed using sparse linear algebra and will have the same sparsity pattern as L_{11}. In the computation of K_{21}, K_{11} is known to be sparse, whereas the density of L_{21} and L_{22} may vary. The K_{21} block can expect a fill-in compared to L_{22}. The K_{22} block will typically be close to half full. See Figure 4, right.

Using the Cholesky factor, the posterior covariance matrix can be computed as

$$C = N^{-1} = (LL^T)^{-1} = K^T K \tag{15a}$$
$$= \begin{pmatrix} K_{11}^T K_{11} + K_{21}^T K_{21} & K_{21}^T K_{22} \\ K_{22}^T K_{21} & K_{22}^T K_{22} \end{pmatrix}. \tag{15b}$$

For notational convenience, let $P = K_{11}$, $Q = K_{21}$. Then,

$$C_{oo} = P^T P + Q^T Q. \tag{16}$$

Note that the K_{22} block is not needed. The structure of the

3-by-3 diagonal blocks of C_{oo} are

$$C_{oo} = \begin{pmatrix} \begin{matrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{matrix} & & \\ & \begin{matrix} c_{44} & c_{45} & c_{46} \\ c_{54} & c_{55} & c_{56} \\ c_{64} & c_{65} & c_{66} \end{matrix} & \\ & & \ddots \end{pmatrix}$$

$$= \begin{pmatrix} \begin{matrix} k_1^T k_1 & k_1^T k_2 & k_1^T k_3 \\ k_2^T k_1 & k_2^T k_2 & k_2^T k_3 \\ k_3^T k_1 & k_3^T k_2 & k_3^T k_3 \end{matrix} & & \\ & \begin{matrix} k_4^T k_4 & k_4^T k_5 & k_4^T k_6 \\ k_5^T k_4 & k_5^T k_5 & k_5^T k_6 \\ k_6^T k_4 & k_6^T k_5 & k_6^T k_6 \end{matrix} & \\ & & \ddots \end{pmatrix}, \tag{17}$$

where k_i are column vectors of K. Thus, the diagonal elements of C_{oo} are the inner products of k_i with themselves. Furthermore, the (2,1) element within each block is the inner product of k_i and k_{i+1}, and similar for the other off-diagonal elements. Due to symmetry, only the sub-diagonal elements need to be computed.

Given that

$$k_i = \begin{pmatrix} p_i \\ q_i \end{pmatrix}, \tag{18}$$

the inner product can be computed as

$$k_i^T k_j = p_i^T p_j + q_i^T q_j, \tag{19}$$

where p_i is sparse with at most three non-zeros, but q_i may be dense. Thus, the computation may be split into a combination of sparse and dense linear algebra.

The computation of the diagonal elements can be performed efficiently in Matlab using a combination of the Hadamard (element-wise) product $\odot$ and the `colsum` function. The call `v=colsum(A)` returns a vector such that v_i holds the sum of the elements of the i:th column of A. Also, let the function `extract(A,d)` return every third column of A starting with column d.

The actual computation is given in Algorithm 2. The diagonal elements are collected in one $3n$-vector d, where n is the number of object points. The sub-diagonal elements are collected in three n-vectors s_{ij}. Finally, the call `C=assemble`$(d, s_{21}, s_{31}, s_{32})$ assembles the vectors to a block-diagonal matrix C.

3.3 Sparse inverse

The SparseInv (SI) algorithm (Förstner and Wrobel, 2016; Kallin, 2019) is a C implementation of the Takahashi et al. (1980) algorithm (Davis, 2014). The SI algorithm computes a subset of the non-zero elements of the inverse $C = N^{-1}$.

3.4 Legacy DBAT

The posterior covariance computation in DBAT up until version 0.9.1 used the inverse Cholesky approach but looped over the object points.

If the normal matrix is in arrowhead ordering and the identity matrix is column blocked as

$$I = \begin{pmatrix} E_1 & E_2 & \dots \end{pmatrix}, \tag{20}$$

Algorithm 2 The inverse Cholesky algorithm for computing posterior covariance for object points.

Require: N has arrowhead ordering
1: $L \leftarrow$ `chol`(N) ▷ Sparse Cholesky factorisation
2: $P \leftarrow L_{11}^{-1}$ ▷ Sparse block inverse
3: $S \leftarrow L_{21} P$ ▷ Sparse multiplication
4: $T \leftarrow$ `full`(L_{22}) ▷ Convert to full
5: $Q \leftarrow -T^{-1} S$ ▷ Dense triangular solve
6: $d \leftarrow$ `colsum`$(P \odot P)$ ▷ Sparse
7: $\quad + $ `colsum`$(Q \odot Q)$ ▷ Dense
8: **for** $i = 1, 2, 3$ **do**
9: $\quad P_i \leftarrow$ `extract`(P, i) ▷ Sparse
10: $\quad Q_i \leftarrow$ `extract`(Q, i) ▷ Dense
11: **end for**
12: **for** $(i, j) = (2, 1), (3, 1), (3, 2)$ **do**
13: $\quad s_{ij} \leftarrow$ `colsum`$(P_i \odot P_j)$ ▷ Sparse
14: $\quad\quad + $ `colsum`$(Q_i \odot Q_j)$ ▷ Dense
15: **end for**
16: $C \leftarrow$ `assemble`$(d, s_{21}, s_{31}, s_{32})$ ▷ Assemble output

Algorithm 3 The legacy DBAT algorithm for computing posterior covariance for object points.

Require: N has arrowhead ordering
1: $L \leftarrow$ `chol`(N) ▷ Sparse Cholesky factorisation
2: **for all** object points i **in** steps of 30 **do**
3: $\quad E \leftarrow$ columns from I.
4: $\quad W \leftarrow L^{-1} E$ ▷ Sparse triangular solve
5: $\quad U \leftarrow W^T W$ ▷ Sparse
6: $\quad$ Make U block diagonal.
7: $\quad$ Insert U in C at the appropriate place.
8: **end for**

where each block column is three columns wide, the block C_{ii} corresponding to object point i becomes (cf. eq. (15a))

$$C_{ii} = E_i^T C E_i = E_i^T N^{-1} E_i = E_i^T K^T K E_i = W_i^T W_i, \tag{21}$$

where $W_i = K E_i$.

In an attempt to use memory more efficiently, the implemented algorithm loops over blocks of 30 points. The algorithm is given as Algorithm 3.

3.5 Total variance only

The diagonal elements of C_{oo} can be used to compute the total variance (trace) of each point. The necessary modification of the classic algorithm is to modify step 8 to compute the diagonal elements only. In the inverse Cholesky algorithm, steps 8–15 are omitted and the matrix is assembled from the diagonal only.

3.6 Self-calibration

For self-calibration projects, the only modification of the classic Algorithm 1 is to have steps 1–2 include the IO blocks, i.e., to match

$$N = \left(\begin{array}{cc|c} N_{ii} & N_{ie} & N_{io} \\ N_{ei} & N_{ee} & N_{eo} \\ \hline N_{oi} & N_{oe} & N_{oo} \end{array}\right) = \left(\begin{array}{c|c} A & B \\ \hline B^T & D \end{array}\right). \tag{22}$$

No modification of the inverse Cholesky algorithm is necessary. Instead the L_{21} and L_{22} blocks are re-interpreted to correspond to all non-OP parameters.

Table 1: Statistics for the data sets used in the paper.

Data Set	Name	Type	Images	Object points	Image points	Ray count IP/OP	Point density IP/Ims	OP/Ims
1	Camcal	Camera calibration	21	100	2074	20.7	99	5
2	Vexcel	Aerial, vertical	41	43251	141510	3.3	3451	1055
3	Roma	Close-range	60	26321	90561	3.4	1509	439
4	StPierre	Close-range	239	17993	196715	10.9	823	75
5	UmU-2D	Drone, vertical strips	133	70600	287528	4.1	2162	531
6	Sewu	Drone, vertical strips	210	107785	658438	6.1	3135	513
7	UmU-3D-R	Drone, vertical cross-hatch	210	107226	477752	4.5	2275	511
8	UmU-3D	Drone, vertical cross-hatch	272	133948	636883	4.8	2341	492
9	UmU-2-LYR	Drone, 2-layer	607	296516	1484959	5.0	2446	488
10	UmU-3-LYR	Drone, 3-layer	944	426971	2290687	5.4	2427	452

4. EXPERIMENTS AND RESULTS

4.1 Data sets

Ten data sets of varying sizes were prepared for the experiments:

Camcal A 21-image camera calibration data set from Börlin and Grussenmeyer (2014).

Vexcel A 41-image vertical aerial dataset flown at an altitude of 950 m above sea level (ASL) above the city of Strasbourg with an UltraCam Osprey Mark 3 Premium camera.

Roma A 60-image terrestrial close-range data set of the *Arco di Constantino* monument in Rome, Italy from Börlin and Grussenmeyer (2013).

StPierre A 239-image close-range data set of the *St-Pierre-le-Jeune* church in Strasbourg from Murtiyoso et al. (2017).

Sewu A 210-image data set acquired by the DJI Phantom 4 RTK drone flown at 60 m ASL over the *Sewu* temple in Central Java, Indonesia.

UmU Five data sets from several flights with a DJI Phantom 4 V2.0 drone over the Umeå University Campus, Umeå, Sweden.

- **UmU-3D** A 272-image data set flown at 70 m ASL in "3D" cross-hatch mode. The data set includes both east-west and north-south strips.
- **UmU-2D** A 133-image subset of UmU-3D with the east-west strips only.
- **UmU-3D-R** The UmU-3D data set, reduced to 210 images to match Sewu. The reduction was done by removing every other north-south strip.
- **UmU-2-lyr** UmU-3D extended with another "3D" flight at 120 m ASL over a larger area.
- **UmU-3-lyr** UmU-2-lyr extended with another "3D" flight at 40 m ASL over a smaller area.

All data sets were self-calibration data sets. The Camcal and Roma images were processed by PhotoModeler Scanner 2012[1] and imported to DBAT. The remaining data sets were processed by AgiSoft Metashape v1.6[2]. During import of the Metashape projects into DBAT, all object points with fewer than 3 rays and/or an estimated intersection angle below 5 degrees were removed before processing by DBAT. The statistics for the data sets are presented in Table 1.

[1] `https://www.photomodeler.com/`
[2] `https://www.agisoft.com/`

4.2 Algorithm variants

The following algorithm variants were implemented and used to process normal matrices:

CC The classic Algorithm 1 of Section 3.1.

SI_1 The sparse inverse algorithm of Section 3.3 with the normal matrix in standard ordering.

SI_2 Same as SI_1, but with the normal matrix in arrowhead ordering.

IC The inverse Cholesky Algorithm 2 of Section 3.2.

IC-S The inverse Cholesky Algorithm 2 where the sparse-to-dense-conversion step 4 was omitted and all subsequent operations were performed using sparse linear algebra.

LD The legacy DBAT Algorithm 3 of Section 3.4.

4.3 Experiments

A total of four experiments were performed:

Self-calibration (SC) The unchanged normal matrices of each data set of Section 4.1 were given to each of the algorithms of Section 4.2.

No self-calibration (NSC) Same as the SC experiment, except the normal matrices were stripped of the IO blocks before processing.

Total variance only (TV) The SC and NSC experiments were repeated with the diagonal-only-variants of the CC and IC algorithm described in Section 3.5.

The effect of the number of cores The SC experiment was repeated with the CC, SI_1, and IC algorithms where the number of available CPU cores were varied from one to six.

In all cases, the execution time for each algorithm on each data set was recorded. The normal matrix was assumed to have been computed by the last iteration of the preceding bundle adjustment. Thus, the time to compute the normal matrix was not included. Furthermore, the time to permute the normal matrix to the ordering preferred by each algorithm was excluded. Additionally, the density of the L_{22} and K_{21} blocks of the IC algorithm were recorded, as well as the density of the K_{22} block.

The timings were performed on an HP Z440 workstation from 2015 with a 6-core Intel Xeon E5-1650 v3 @ 3.50GHz and 64GB of RAM running Matlab 2019b Update 1 under Debian 10.

Table 2: Block non-zero density for the data sets. The B column correspond to the B block of the original problem in equations (22) and (5), respectively. The K_{21} (Q), and L_{22} columns are densities for corresponding blocks in the IC algorithm. All recorded K_{22} densities were very close to 50%.

		Self-calibration			No self-calibration		
Data Set	Name	B	K_{21}	L_{22}	B	K_{21}	L_{22}
1	Camcal	98.8	100.0	50.4	98.7	100.0	50.4
2	Vexcel	10.9	62.8	44.0	8.0	61.6	43.5
3	Roma	8.0	54.8	31.0	5.7	53.6	30.0
4	StPierre	5.1	66.6	47.4	4.6	66.5	47.4
5	UmU-2D	4.0	55.3	31.8	3.1	54.9	31.4
6	Sewu	3.5	55.6	33.6	2.9	55.3	33.4
7	UmU-3D-R	2.7	62.4	38.0	2.1	62.1	37.8
8	UmU-3D	2.2	63.8	41.0	1.7	63.6	40.9
9	UmU-2-LYR	1.0	58.5	43.4	0.8	58.4	43.3
10	UmU-3-LYR	0.7	59.2	45.4	0.6	59.2	45.4

Table 3: Execution times in seconds for the SC, NSC, and TV experiments. The fastest algorithm within each block is highlighted. Overall, the IC algorithm is the fastest for small data sets, whereas the CC algorithm is fastest for large data sets. The SI algorithm has intermediate performance. The cross-over point depends on the requested computation and whether the data set is a self-calibration data set or not. The IC algorithm is faster than the legacy LD algorithm by 1–2 orders of magnitude.

		Full 3-by-3 blocks								Diagonal only			
Data		Self-calibration				No self-calibration				Self-cal.		No self-cal.	
Set	Name	CC	SI	IC	LD	CC	SI	IC	LD	CC	IC	CC	IC
1	Camcal	0.1	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	Vexcel	1.9	2.7	0.7	15.6	1.1	1.6	0.5	15.6	1.1	0.5	0.7	0.4
3	Roma	1.3	1.7	0.5	9.4	0.7	1.0	0.4	9.3	0.8	0.4	0.4	0.3
4	StPierre	9.9	6.9	1.3	55.3	8.2	6.4	1.2	55.4	5.2	1.1	4.6	1.0
5	UmU-2D	4.0	6.1	2.1	90.9	2.5	3.6	2.0	91.0	2.4	1.6	1.5	1.5
6	Sewu	12.8	14.8	5.4	326.4	9.6	11.0	5.2	327.2	7.5	4.2	5.7	4.0
7	UmU-3D-R	7.3	11.7	5.1	314.5	4.9	9.8	4.9	314.5	4.3	3.9	2.9	3.7
8	UmU-3D	10.4	16.9	8.8	623.3	7.2	13.0	8.5	623.4	6.1	6.9	4.3	6.6
9	UmU-2-LYR	27.2	47.6	62.6	6443.6	19.8	40.2	62.3	6422.0	16.2	53.8	11.9	53.5
10	UmU-3-LYR	48.3	99.3	191.9	22877.1	37.0	88.4	189.6	22663.9	28.9	172.1	22.5	170.4

4.4 Results

The density numbers are given in Table 2. The L_{22} density varied between 30% and 50%. Except for the camera calibration data set, the K_{21} density varied between 53% and 66%. All K_{22} densities were very close to 50%.

The execution times for the CC, SI_1, IC, and LD algorithms for the SC and NSC experiment are given in Table 3. The CC execution times were dominated (80–90%) by the computation of the 3-by-3 blocks in step 8. For the large problems (data sets 6–10), the IC execution times were dominated by (50–85%) by the K_{21} (Q) computation in step 5. The execution times for the SI_2 algorithm was within 15% of the SI_1 algorithm. The IC-S algorithm was several orders of magnitude slower than the IC algorithm. The results show that the IC algorithm is fastest on SC data sets 1–8 and on NSC data sets 1–7. The CC algorithm is fastest on the remaining data sets, with the SI algorithm somewhere in between. In both cases, the cross-over point correspond to a B density of about 2%. Except for the camera calibration data set, the IC algorithm is 20-120 times faster than the legacy LD algorithm. For the TV experiment, also presented in Table 3, the IC algorithm was fastest for SC data sets 1–7 and NSC data sets 1–6. In this case, the cross-over point corresponds to a B density of about 2.5%.

On average, the execution times for the SC data sets were 40–50% higher for the CC and SI algorithms compared to their NSC counterparts. In contrast, with the exception of the camera calibration data set, the SC and NSC times were within 1% of each other. In the diagonal-only TV experiment, the execution times decreased by about 40% for the CC algorithm and about 20% for the IC algorithm compare the full-block SC and NSC results.

The Sewu and UmU-3D-R data sets has the same number of images and object points. However, the number of image points are 38% higher in the Sewu data set. The same difference is seen in the number of rays per object point and the B-block density for the NSC data set, with a slightly lower difference for the SC data set. The increased point density translated to a 75-100% longer execution time for the CC algorithm. In contrast, the corresponding increase for the IC algorithm was about 6%.

The result of the speedup Experiment 4 are shown in Figure 5. Only the IC algorithm benefited from an increased number of cores, but with a decreasing efficiency as the number of cores are increased. Using six cores the speedup was about 2.5. On a single core, the IC algorithm was faster the the CC algorithm for data sets 1–4, and within a factor of 2 for data sets 1–8.

5. DISCUSSION

In this paper, several algorithm for the computation of posterior covariance of object points were applied to self-calibration and non-self-calibration data sets of varying sizes, ranging from 21-944 images. On the smallest data sets, the fastest algorithm was the Inverse Cholesky (IC) algorithm. On the largest data sets, the classic (CC) algorithm was fastest. The performance of the Sparse Inverse (SI) algorithm was somewhere in between CC and IC. The relative results between SI and IC are consistent with the results by Kallin (2019), on similar problem sizes.

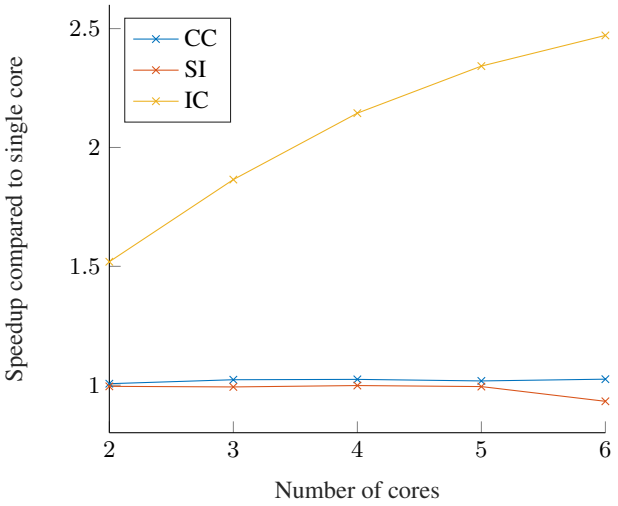

Figure 5: Average speedup as a function of the number of cores. Only the IC algorithm benefits from an increased number of cores, but at a diminishing rate as the number of available cores increases.

Compared to the legacy algorithm in DBAT, the IC algorithm was 1-2 orders of magnitude faster.

Several factors affect the computation time for a given algorithm, including the number of images, object points, and image points. A critical parameter appears to be the density of the mixed block of the normal matrix. As the image and point numbers increase, the density of the mixed block typically decreases. When the full 3-by-3 blocks of the posterior covariance matrix are computed, the density cross-over point for the fastest algorithm appear to be around 2%.

If the posterior correlations of the estimated object points are needed, the full 3-by-3 block must be computed. Otherwise, if only the total variance of each point is needed, only the diagonal elements of the posterior covariance have to be computed. In this case, a decrease in execution time of about 40% for the CC algorithm and 20% for the IC algorithm was observed, compared to the full block computations and the density cross-over point shifted to about 2.5%.

The IC algorithm is written completely in the Matlab language. As such, it benefits from an increased number of available cores due to the efficiency of the underlying numeric libraries used by Matlab. The performance increase was sub-linear. In contrast, the SI and CC algorithms were partially implemented in the C language and called from Matlab. The C code was not optimised for multiple cores. Further performance improvements are likely if the C code is parallellised, although the level is uncertain.

For a general toolbox as DBAT, a pure Matlab implementation is advantageous, as it removes any dependencies on, e.g., compilers. However, for a specific lab with mostly large projects, compiling and using the classic algorithm will give improved performance.

Compared to the previous version of DBAT, users can expect posterior covariance computations that are a few orders of magnitude faster. In summary, the computation time for the largest, 944-image dataset in this paper decreased by a factor of over 100. The actual computation time was around three minutes.

6. CONCLUSIONS

On medium-sized data sets of up to about 300 images, the Matlab-only IC algorithm is faster than the classic algorithm. On larger data sets, the classic algorithm is faster. However, on data sets up to about 900 images, and if a computation time of a few minutes at the end of a bundle adjustment processing is acceptable, the IC algorithm is still a useful high-level alternative.

We have shown throughout this paper that the novel IC algorithm implemented in DBAT managed to accelerate its computation of covariance values significantly. Experiments by comparing it to other methods, including the classical approach, showed that the results are comparable, albeit with a few caveats. For DBAT this is a significant addition as it enables a relatively rapid computation of more statistical metrics, further reinforcing its application for reprocessing bundle adjustment results of black-box solutions. Further investigations and experiments in this regard may also reveal other options for optimisation.

References

Börlin, N., Grussenmeyer, P., 2013. Bundle Adjustment with and without Damping. *Photogrammetric Record*, 28(144), 396-415.

Börlin, N., Grussenmeyer, P., 2014. Camera Calibration using the Damped Bundle Adjustment Toolbox. *ISPRS Annals of the Photogrammetry, Remote Sensing, and Spatial Information Sciences*, II(5), 89-96. Best paper award.

Börlin, N., Murtiyoso, A., Grussenmeyer, P., Menna, F., Nocerino, E., 2019. Flexible Photogrammetric Computations using Modular Bundle Adjustment. *Photogrammetric Engineering and Remote Sensing*, 85(5), 361-368.

Brown, D. C., 1968. Inversion of very large matrices encountered in large scale problems of photogrammetry and photographic astrometry. H. K. Eichhorn (ed.), *Proceedings of Conference on Photographic Astrometric Technique*, NASA, Department of Astronomy, University of South Florida, Tampa, Florida, 249–267. Also in NASA CR-1825 published November 1971.

Brown, D. C., 1976. The Bundle Adjustment — Progress and prospects. *International Archives of Photogrammetry, Remote Sensing, and Spatial Information Sciences*, 21(3), 33 pp.

Davis, T., 2014. Sparseinv: Sparse inverse subset. MATLAB Central File Exchange. https://www.mathworks.com/matlabcentral/fileexchange/33966-sparseinv-sparse-inverse-subset). Retrieved May 3, 2020.

Duff, I. S., Heroux, M. A., Pozo, R., 2002. An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum. *ACM Trans. Math. Softw.*, 28(2), 239–267. https://doi.org/10.1145/567806.567810.

Förstner, W., Wrobel, B. P., 2016. *Photogrammetric Computer Vision*. Springer.

Goto, K., Van De Geijn, R., 2008. High-Performance Implementation of the Level-3 BLAS. *ACM Trans. Math. Softw.*, 35(1). https://doi.org/10.1145/1377603.1377607.

Ila, V., Polok, L., Solony, M., Istenic, K., 2017. Fast incremental bundle adjustment with covariance recovery. *2017 International Conference on 3D Vision (3DV)*, 175–184.

K. Takahashi, K., Fagan, J., Chen, M.-S., 1980. Formation of a Sparse Bus Impedance Matrix and its Application to Short Circuit Study. *IEEE Power Engineering Society*, 7(3), 16-29. Also in PICA Conference, 1973, Abstract TPIIB in trans. IEEE. PAS-93 Jan.–Feb. (1974) 3, 1A06.

Kallin, N., 2019. Computation of posterior covariances of object points in bundle adjustment. Master's thesis, Umeå University, Department of Computing Science. Report UMNAD 1179.

Kraus, K., 2007. *Photogrammetry*. 1, 2 edn, de Gruyter, Berlin.

Luhmann, T., Robson, S., Kyle, S., Boehm, J., 2014. *Close-Range Photogrammetry and 3D Imaging*. 2nd edn, De Gruyter, Berlin, Germany.

Murtiyoso, A., Grussenmeyer, P., Börlin, N., 2017. Reprocessing Close Range Terrestrial and UAV Photogrammetric Projects with the DBAT Toolbox for Independent Verification and Quality Control. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W8, 171–177.

Murtiyoso, A., Grussenmeyer, P., Börlin, N., Vandermeerschen, J., Freville, T., 2018. Open Source and Independent Methods for Bundle Adjustment Assessment in Close-Range UAV Photogrammetry. *Drones*, 2(1). http://www.mdpi.com/2504-446X/2/1/3.

Triggs, B., McLauchlan, P., Hartley, R., Fitzgibbon, A., 2000. Bundle adjustment — A modern synthesis. *Vision Algorithms: Theory and Practice, Proceedings of the International Workshop on Vision Algorithms*, Lecture Notes in Computer Science, 1883, Springer Verlag, 298–372.

Wolf, P., DeWitt, B., Wilkinson, B., 2013. *Elements of Photogrammetry with Application in GIS*. 4 edn, McGraw-Hill.