# Incremental Block Cholesky Factorization for Nonlinear Least Squares in Robotics

Lukas Polok, Viorela Ila, Marek Solony, Pavel Smrz and Pavel Zemcik
Brno University of Technology, Faculty of Information Technology.
Bozetechova 2, 612 66 Brno, Czech Republic ({ipolok,ila,isolony,smrz,zemcik}@fit.vutbr.cz)

***Abstract*—Efficiently solving nonlinear least squares (NLS) problems is crucial for many applications in robotics. In online applications, solving the associated nolinear systems every step may become very expensive. This paper introduces online, incremental solutions, which take full advantage of the sparse-block structure of the problems in robotics. In general, the solution of the nonlinear system is approximated by incrementally solving a series of linearized problems. The most computationally demanding part is to assemble and solve the linearized system at each iteration. In our solution, this is mitigated by incrementally updating the factorized form of the linear system and changing the linearization point only if needed. The incremental updates are done using a resumed factorization only on the parts affected by the new information added to the system at every step. The sparsity of the factorized form directly affects the efficiency. In order to obtain an incremental factorization with persistent reduced fill-in, a new incremental ordering scheme is proposed. Furthermore, the implementation exploits the block structure of the problems and offers efficient solutions to manipulate block matrices, including a highly efficient Cholesky factorization on sparse block matrices. In this work, we focus our efforts on testing the method on SLAM applications, but the applicability of the technique remains general. The experimental results show that our implementation outperforms the state of the art SLAM implementations on all the tested datasets.**

## I. Introduction

Many applications ranging from robotics, computer graphics, computer vision to physics rely on efficiently solving large nonlinear least squares (NLS) problems. The goal is to find the optimal configuration of a set of variables that maximally satisfy the set of soft nonlinear constraints. For instance in robotics, simultaneous localization and mapping (SLAM) estimates the optimal configuration of the robot positions and/or landmarks in the environment given a set of sensor measurements [6, 7, 10, 11, 14]. The variables can have several degrees of freedom (e.g. 3DOF for 2D problems or 6DOF for 3D problems). Therefore, the associated system matrix can be interpreted as partitioned into sections corresponding to each variable, called *blocks*, which can be manipulated at once.

In practice, the initial problem is nonlinear and it is usually addressed by repeatedly solving a sequence of linear systems. The linear system can be solved either by matrix factorization or gradient methods. The latter are more efficient from the storage point of view, since they only require access to the gradient, but they can suffer from poor convergence, slowing down the execution. Matrix factorization, on the other hand, produces more accurate solutions and avoids convergence difficulties but typically requires a lot of storage.

The problems in robotics are in general *sparse*, which means that the associated system matrix is primarily populated with zeros. Many efforts have been recently made to develop efficient methods to store and manipulate sparse matrices. CSparse [3], developed by Tim Davis [4] is one of the most used sparse linear algebra libraries. It is highly optimized in terms of run time and memory storage and it is also very easy to use. CSparse stores the sparse matrices in compressed column format (CCS) which considerably reduces the memory requirements and is suitable for matrix operations.

The block structure and the sparsity of the matrices can bring important advantages in terms of storage and matrix manipulation. Some of the existing implementations rely on sparse block structure schemes [14, 13]. In the existing schemes, the block structure is maintained until the point of solving the linear system. Here is where CSparse [4] or CHOLMOD [5] libraries are used to perform the element-wise matrix factorization. Once it has been compressed, it becomes impractical and inefficient to change a matrix structurally or numerically. This motivated us to find efficient solutions to matrix storage and operations on matrices, especially the matrix factorization, to avoid the need of converting from blockwise to elementwise CCS format and back.

In online applications the state changes every step. In SLAM, for example, the state is *augmented* with a new robot position or a new landmark from the environment and it is *updated* with the corresponding measurement. Every iteration of the nonlinear solver involves building a new linear system using the current linearization point and solving it using matrix factorization. For very large problems, updating and solving the nonlinear system every step can become very expensive. The literature proposes elegant incremental solutions, either by working directly on the matrix factorization [10] or by using a graphical model-based data structure called the Bayes tree, which allows efficient incremental algorithms [12, 11]. In their early work, Kaess et al. [10] introduced the idea of keeping the factorized form of the linear system and only recalculating parts that are affected by the incremental updates. This method becomes advantageous when the *batch steps* are performed periodically. The batch steps involve changing the linearization point by solving the nonlinear system and are needed for two important reasons; a) the error increases if the same linearization point is kept for a long time and b) the fill-in of the factorized form increases with the incremental updates, slowing down the backsubstitution. Later, they introduced the

Bayes tree data structure [12], which provides insights on the connection between graphical model inference and sparse matrix factorization. This offered the possibility of eliminating the need for periodic batch steps obtaining incremental variable re-ordering to reduce the fill-in and fluid relinearization to guarantee good linearization points [11].

The work introduced in this paper combines the efficiency of operating directly on the matrix factorization with the insights gained from the Bayes tree data structure to produce highly efficient incremental solutions. Our incremental solution is based on directly updating the Cholesky factorization of the linear system and changing the linearization point only if the error increases. This guarantees both fast and high quality estimations. The proposed algorithm is based on a resumed Cholesky factorization which recalculates only the parts affected by the new updates, together with an incremental reordering scheme which maintains the factorization sparse without the need of periodic batch steps.

Our implementation maximally exploits the sparse block structure of the problem. On one hand, the block matrix manipulation is highly optimized, facilitating structural and numerical matrix changes while also performing arithmetic operations efficiently. On the other hand, the block structure is maintained in all the operations including the matrix factorization, eliminating the cost of converting between sparse elementwise and sparse blockwise. Our block Cholesky factorization implementation proves to be significantly faster than the existing state of the art elementwise implementations.

The contributions of this work are introduced as follows. The next section succinctly formalizes SLAM as a nonlinear least squares problem. Incremental updates are described in Section III. Then, Section IV explains the details of the proposed algorithm and implementation. In the experimental evaluation in Section V we show the increased efficiency of our proposed scheme over the existing implementations. Conclusions and future work are given in Section VI.

## II. Nonlinear Least Squares Problem in Robotics

In robotics, simultaneously localizing a robot and mapping the environment is often formulated as a nonlinear least squares problem (NLS) [6]. The goal is to obtain the maximum likelihood estimate (MLE) of a set of variables $\boldsymbol{\theta} = [\theta_1 \dots \theta_n]$, usually containing the robot trajectory and/or the position of landmarks in the surrounding environment, given the set of measurements $\mathbf{z}$:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\ P(\boldsymbol{\theta} \mid \mathbf{z}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\ \{-\log(P(\boldsymbol{\theta} \mid \mathbf{z})\} \ . \quad (1)$$

For every measurement $z_k = h_k(\theta_{i_k}, \theta_{j_k}) - v_k$ we assume the Gaussian distribution:

$$P(z_k \mid \theta_{i_k}, \theta_{j_k}) \quad \propto \quad \exp\left(-\frac{1}{2} \parallel h_k(\theta_{i_k}, \theta_{j_k}) - z_k \parallel^2_{\Sigma_k}\right) \quad (2)$$

where $h(\theta_{i_k}, \theta_{j_k})$ is the nonlinear measurement function, and where $v_k$ is the normally distributed zero-mean noise with the covariance $\Sigma_k$. Finding the MLE from (1) is done by solving the following nonlinear least squares problem:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \left\{ \frac{1}{2} \sum_{k=1}^{m} \| h_k(\theta_{i_k}, \theta_{j_k}) - z_k \|^2_{\Sigma_k} \right\} . \quad (3)$$

Methods such as Gauss-Newton or Levenberg-Marquardt are often used to solve the NLS in (3) and this is usually addressed by iteratively solving the sequence of linear systems. Those are obtained by linear approximations of the nonlinear residual functions around the current linearization point $\boldsymbol{\theta}^i$:

$$\tilde{\mathbf{r}}(\boldsymbol{\theta}^i) = \mathbf{r}(\boldsymbol{\theta}^i) + J(\boldsymbol{\theta}^i)(\boldsymbol{\theta} - \boldsymbol{\theta}^i) \ , \quad (4)$$

where $\mathbf{r}(\boldsymbol{\theta}) = [r_1, \dots, r_m]^\top$ is the set of all nonlinear residuals of type $r_k = h_k(\theta_{i_k}, \theta_{j_k}) - z_k$ and $J$ is the Jacobian matrix which gathers the derivative of the components of $\mathbf{r}(\boldsymbol{\theta}^i)$. The linearized problem to solve becomes:

$$\boldsymbol{\delta}^* = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \frac{1}{2} \| A\,\boldsymbol{\delta} - \mathbf{b} \|^2 \ , \quad (5)$$

where the $A = \boldsymbol{\Sigma}^{-\top \backslash 2} J$ is the system matrix, $\mathbf{b} = -\mathbf{r}(\boldsymbol{\theta}^i)$ the right hand side (r.h.s.) and $\boldsymbol{\delta} = (\boldsymbol{\theta} - \boldsymbol{\theta}^i)$ the correction to be calculated [6]. This is a standard linear least squares problem in $\boldsymbol{\delta}$. For SLAM problems, the matrix $A$ is in general sparse, since every measurement depends only on a few variables, but it can become very large when the robot performs long trajectories. The normal system equation has the advantage of remaining of the size of the state even if the number of measurements increases:

$$\boldsymbol{\delta}^* = \underset{\boldsymbol{\delta}}{\operatorname{argmin}} \frac{1}{2} \| \Lambda\boldsymbol{\delta} - \boldsymbol{\eta} \|^2 \ , \quad (6)$$

where $\Lambda = A^\top A$ is the information matrix and $\boldsymbol{\eta} = A^\top \mathbf{b}$ is the information vector.

The solution of the linear system can be obtained by sparse matrix factorization followed by backsubstitution. In general, QR factorization of $A$ has slightly better numerical properties, but Cholesky factorization on $\Lambda$ performs much faster. In this paper we are interested in obtaining very fast online solutions, therefore we apply Cholesky factorization.

The Cholesky factorization of a square symmetric positive definite matrix $\Lambda$ has the form $L\,L^\top = \Lambda$ , where $L$ is a lower triangular matrix with positive diagonal entries. The forward and backsubstitutions on $L\mathbf{d} = A^\top \mathbf{b}$ and $L^\top\,\boldsymbol{\delta} = \mathbf{d}$ first recovers $\mathbf{d}$, then the actual solution $\boldsymbol{\delta}$. After computing $\boldsymbol{\delta}$, the new linearization point becomes $\boldsymbol{\theta}^{i+1} = \boldsymbol{\theta}^i \oplus \boldsymbol{\delta}$, where $\oplus$ is the exponential map operator [14]. The Gauss-Newton iterates until the norm of the correction becomes smaller than a tolerance.

## III. Incremental System Updates

In online robotic applications such as SLAM, new variables (robot poses and/or landmarks) are integrated every step and new measurements frequently update the system. Integrating a new variable involve *augmenting* the system with the size of the new variable and *updating* it with the corresponding measurement. Integrating a measurement may become expensive, without carefully performing the involved operations.

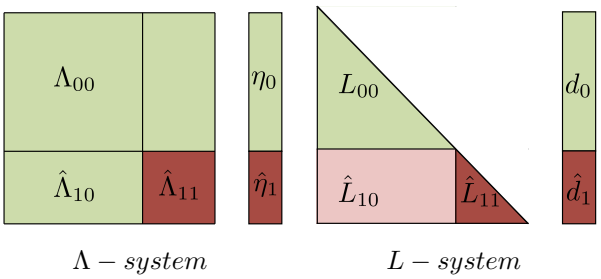


Fig. 1. Incremental updates on $\Lambda$-system and $L$-system. The blocks are color-coded according to the way they are affected by the update. Green - NOT affected, red - affected, pink - affected by performing resumed factorization.

### A. Incrementally updating the system matrix

Updating the system with a new measurement is additive in information form. We denote $\Omega = H^\top \Sigma_k^{-1} H$ and $\boldsymbol{\omega} = -H\Sigma_k^{-1\backslash 2} r_k$ to be the increments in information, where $H$ is the Jacobian of the new measurement. In general, in SLAM, the measurement function $h_k(\cdot)$ involves only two variables, $(\theta_i, \theta_j)$. For this reason and for simplicity, the following formulation will be restricted to measurements between two variables but its application remains general. The corresponding Jacobian, $H$, is very sparse $H = \begin{bmatrix} 0 \dots J_i^j \dots 0 \dots J_j^i \end{bmatrix}$ and this translates into a sparse $\Omega$ and $\boldsymbol{\omega}$. The update step only partially changes the information matrix $\Lambda$ and the information vector $\boldsymbol{\eta}$. For simplicity of the notations, in the following formulations, the system matrices are split in parts that change ($\Lambda_{11}$, $\boldsymbol{\eta}_1$) and parts that remain unchanged ($\Lambda_{00}$, $\Lambda_{10}$ and $\boldsymbol{\eta}_0$):

$$\hat{\Lambda} = \begin{bmatrix} \Lambda_{00} & \Lambda_{10}^\top \\ \Lambda_{10} & \Lambda_{11} + \Omega \end{bmatrix}; \; \hat{\boldsymbol{\eta}} = \begin{bmatrix} \boldsymbol{\eta}_0 \\ \boldsymbol{\eta}_1 + \boldsymbol{\omega} \end{bmatrix}, \tag{7}$$

In the formulation above we deliberately considered that the current measurement to be integrated involves the last variable added to the system. This is the situation usually encountered in incremental SLAM problem. Note that this assumption is not necessarily needed, the formulation in (7) stays general. Fig. 1 illustrates the unaffected part of the system matrix and r.h.s in green and the affected in red. When the measurement involves a new variable, the state is augmented with a zero block-row and a zero block-column (block-size depending on the size of the new variable) and updated with the corresponding measurement. This is an inexpensive process since the new variable always links to variables recently added to the system, but the update step, in general, can become very expensive if the new measurement links variables far apart.

### B. Incrementally updating the Cholesky factor

As shown above, only a small part of the information matrix and the information vector are changed in the update process and the same happens with its factorized form $L$. The updated $\hat{L}$ factor and the corresponding r.h.s. $\hat{\mathbf{d}}$ can be written as:

$$\hat{L} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & \hat{L}_{11} \end{bmatrix} \; ; \quad \hat{\mathbf{d}} = \begin{bmatrix} \mathbf{d_0} \\ \hat{\mathbf{d_1}} \end{bmatrix} . \tag{8}$$

From $\hat{\Lambda} = \hat{L}\,\hat{L}^\top$, (7) and (8) the updated part of the Cholesky factor and the r.h.s can be easily computed:

$$\hat{L}_{11} = chol(L_{11}\, L_{11}^\top + \Omega) \tag{9}$$

$$\hat{\mathbf{d_1}} = \hat{L}_{11} \backslash (\hat{\boldsymbol{\eta}}_1 - L_{10}\, \mathbf{d_0}) \, . \tag{10}$$

The part unaffected by the update is shown in green in the Fig. 1, and the affected parts are shown in red and pink. Variable ordering plays an important role in the matrix factorization, and this directly influences the fill-in of the $L$ factor. Later in the paper it is shown that, in order to maintain the factorization sparse, we will recalculate both, $L_{10}$ and $L_{11}$ using a resumed Cholesky factorization applied to a reordered $\hat{\Lambda}$. This is illustrated in Fig. 1 using pink color for $L_{10}$.

This form of incrementally updating the Cholesky factor is very similar to the incremental updates proposed in [10], where the authors use Givens rotations to update $R = L^\top$ and $d$. Even if this is one of the fast incremental approaches, there are still two important problems. Firstly, without periodic reorderings, the factorized form becomes less and less sparse, slowing down the solving. Another problem is that within an iterative nonlinear solver the linearization point can change every iteration, invalidating the entire factorization. The recently introduced data structure, the Bayes tree [12], offers the possibility to develop incremental algorithms where reordering and re-linearization are performed fluidly, without the need of periodic updates. Inspired by the above-mentioned incremental strategies, this paper proposes an elegant and highly efficient solution which combines the efficiency of matrix implementation and considers the insights gained using the Bayes tree data structure.

## IV. Incremental Block-Cholesky Factorization

This section will discuss several aspects to be considered in the proposed incremental solution such as the block structure of the targeted problems and the importance of the incremental reordering; then, it will describe the block-factorization itself and will conclude proposing an efficient incremental algorithm for the SLAM problem.

### A. Block Matrix Scheme

SLAM, and similar problems such as structure from motion (SfM) and bundle adjustment (BA), involve operations with matrices having a block structure, where the size of the blocks corresponds to the number of degrees of freedom of the variables. Sparsity of such problems plays an important role, therefore, sparse linear algebra libraries such as CSparse [4] or CHOLMOD [5] are commonly used to perform the matrix factorization. Those are state of the art element-wise implementations of operations on sparse matrices. The *element-wise* sparse matrix schemes provide efficient ways to store the sparse data in the memory and perform arithmetic operations. The disadvantage is their inability or impracticality to change matrix structurally or numerically once it has been compressed. The *block-wise* schemes are complementary, their advantages include both, easy numeric and structural matrix

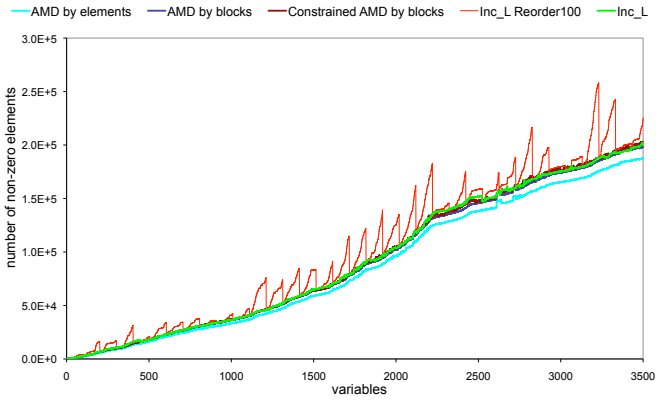


Fig. 2. The comparison in terms of nonzero elements of several ordering heuristics and the actual number of non-zero elements in Incremental L.

modification, at the cost of small memory overhead, and slightly worse arithmetic efficiency.

Our implementation combines the advantages of *block-wise* schemes convenient in both, numeric and structural matrix modification and *element-wise*, which allows fast arithmetic operation on sparse matrices. This is achieved by a careful design of the block matrix storage for cache efficiency in combination with the use of vectorized SSE instructions [**?** ].

### B. Incremental Ordering

The fill-in of the factor $L$ directly affects the speed of the backsubstitution and the updates. Its sparsity depends on the order of the rows and columns of the matrix $\Lambda$, called *variable ordering*. Unfortunately, finding an ordering which minimizes the fill-in of $L$ is NP-complete. Therefore, heuristics have been proposed in the literature [1] to reduce the fill-in of the result of the matrix factorization. In our implementation we use constrained AMD ordering, available as a part of SuiteSparse family of libraries [4].

In an incremental SLAM process, the new variable, either the next observed landmark or the next robot pose, is always linked to an existing pose in the representation. In order to be able to perform efficient incremental updates on the Cholesky factor, the last pose is constrained to be ordered last. Figure 2 shows that the constrained ordering barely affect the fill-in. It also illustrates in red how the fill-in rises when the reordering is performed only every 100 steps. Furthermore, due to the inherent block structure, and in order to facilitate further incremental updates, the ordering is done by blocks. Figure 2 shows that applying ordering by blocks instead of element-wise has very small influence on the fill-in of the $L$ factor. This influence is caused mostly by the fact that the diagonal blocks in $L$ are half empty, but still have to be stored as full blocks.

The work in [12] highlights the importance of reordering the variables every step, as well as offers an elegant solution based on their Bayes tree data structure. Guided by these insights, in this paper we propose a scheme that facilitates efficient partial reordering in the incremental block matrix factorization. Figure 2 shows that this strategy is efficient in terms of factor sparsity, but it runs much faster, compared to full reordering. As mentioned before, the reordering is done on the system matrix $\Lambda$, therefore the proposed method keeps the $\Lambda$ matrix up to date. The matrix is sparse and the updates are additive, therefore this additional process is inexpensive. In Section IV-E we show how this is used in an incremental ordering process assuring a reduced fill-in of the matrix factorization.

### C. Block Matrix Factorization

The Cholesky factorization algorithm is a fast way of computing decompositions in form $A = LL^\top$. It is based on two simple operations, *cdiv* and *cmod*. Cdiv comprises division of a row or a column of the factor by a square root of its diagonal entry. Cmod modifies a row or a column by a multiple of the previous row or column. The $L$ factor is obtained by repeatedly applying those two steps on $\Lambda$ matrix. Cholesky factorization, unlike other factorizations, does not require any pivoting for numerical stability. Sparse Cholesky factorization, however, relies on symbolic ordering of the matrix. If the matrix is not ordered properly, the cmod steps introduce fill-in.

The current element-wise implementations of the Cholesky factorization such as CSparse and CHOLMOD are, in general, based on four steps: reordering, calculating the elimination tree, symbolic factorization and numeric factorization. The elimination tree captures the dependencies between the individual columns or rows of the matrix. It is also the input to the symbolic factorization, which is required for calculation of the nonzero pattern of the factor, in order to be able to produce it in a sparse compressed matrix. CSparse relies on row-Cholesky factorization, which produces one row of the result matrix at a time. At each row, nonzero pattern is computed from the elimination tree. CSparse keeps a dense vector which is loaded with a row and then the cmod operation is executed on that. This approach is not suitable for block Cholesky factorization, as storing one block row as a dense matrix would require substantial amounts of memory.

In our implementation, block Cholesky factorization is employed. The result of this factorization is numerically equivalent to elementwise sparse factorization of the same matrix. The general algorithm is similar, with the only difference that in cdiv step, the division is replaced by backsubstitution and the square root is replaced by Cholesky factorization of a dense block. Since the block matrix $\Lambda$ is symmetric, the diagonal blocks are guaranteed to be square and this factorization is therefore well defined. Similarly, in cmod, the scalar multiplication is replaced by matrix multiplication. Unlike elementwise sparse Cholesky factorization, blockwise factorization does not require symbolic factorization as the matrix can be conveniently changed both structurally and numerically.

### D. Incremental SLAM Algorithm

Our approach to incremental SLAM is described by the pseudocode in Alg. 1, which can be seen as having three

INCREMENTAL-L($\boldsymbol{\theta}, \mathbf{r}, \boldsymbol{\Sigma}_k, L, \Lambda, \boldsymbol{\eta}, \mathbf{o}, maxIT, tol, new_{LP}$)

1: $(\Omega, \boldsymbol{\omega})$ $\leftarrow$ COMPUTEOMEGA($(\theta_i, \theta_j)$ , $r_k$ , $\Sigma_k$)
2: **if** $new_{LP}$ **then**
3: $(\hat{\Lambda}, \hat{\boldsymbol{\eta}})$ $\leftarrow$ LINEARSYSTEM($\boldsymbol{\theta}$ , $\mathbf{r}$)
4: **else**
5: $(\hat{\Lambda}, \hat{\boldsymbol{\eta}})$ $\leftarrow$ UPDATELINEARSYSTEM($\Lambda, \eta, \Omega, \boldsymbol{\omega}$)
6: **end if**
7: $(\hat{L}, \hat{\mathbf{d}}, \hat{\mathbf{o}})$ $\leftarrow$ UPDATE-L($i, j, \Omega, \boldsymbol{\omega}, L, \Lambda, \boldsymbol{\eta}, \mathbf{o}, new_{LP}$)
8: $new_{LP}$ $\leftarrow$ FALSE
9: **if** $maxIT \leq 0 \parallel \neg\, hadLoop$ **then**
10: **exit**
11: **end if**
12: $it = 0$
13: **while** $it < maxIT$ **do**
14: **if** $it > 0$ **then**
15: $(\hat{\Lambda}, \hat{\boldsymbol{\eta}})$ $\leftarrow$ LINEARSYSTEM($\boldsymbol{\theta}$ , $\mathbf{r}$)
16: $(\hat{L}, \hat{\mathbf{d}}, \hat{\mathbf{o}})$ $\leftarrow$ UPDATE-L($i, j, \Omega, \boldsymbol{\omega}, L, \Lambda, \boldsymbol{\eta}, \mathbf{o}, new_{LP}$)
17: $new_{LP}$ $\leftarrow$ FALSE
18: **end if**
19: $\boldsymbol{\delta}$ $\leftarrow$ LSOLVE($\hat{L}$ , $\hat{\mathbf{d}}$)
20: **if** $norm(\boldsymbol{\delta}) \geq tol$ **then**
21: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} \oplus \boldsymbol{\delta}$
22: $new_{LP}$ $\leftarrow$ TRUE
23: **else**
24: **exit**
25: **end if**
26: $it++$
27: **end while**

**Algorithm 1:** Incremental SLAM algorithm.

$(\hat{L}, \hat{\mathbf{d}}, \hat{\mathbf{o}})$ $\leftarrow$ UPDATE-L($i, j, \Omega, \boldsymbol{\omega}, L, \hat{\Lambda}, \hat{\boldsymbol{\eta}}, \mathbf{o}, new_{LP}$)

1: **if** $new_{LP} \parallel$ COLUMNS($\Omega$) = COLUMNS($\hat{\Lambda}$) **then**
2: $\hat{\mathbf{o}} \leftarrow$ CAMD($\hat{\Lambda}$)
3: $\hat{L} \leftarrow$ CHOL($\hat{\Lambda}, \mathbf{o}$)
4: $\hat{\mathbf{d}} \leftarrow$ LSOLVE($\hat{L}$ , $\hat{\boldsymbol{\eta}}$)
5: **else**
6: $o_{lo} \leftarrow$ MIN($\mathbf{o}[i], \mathbf{o}[j]$)
7: $o_{hi} \leftarrow$ COLUMNS($\Lambda$)
8: $\hat{\Lambda}_p \leftarrow$ PERMUTE($\hat{\Lambda}, \mathbf{o}$)
9: $o_{cut} \leftarrow$ MIN(WAVEFRONT($\hat{\Lambda}_p$)$[o_{lo} : o_{hi}]$)
10: $\hat{\Lambda}_{p_{11}} \leftarrow \hat{\Lambda}_p[o_{cut} : o_{hi}, o_{cut} : o_{hi}]$
11: $\mathbf{o_{new}} \leftarrow$ CAMD($\hat{\Lambda}_{p_{11}}$)$[o_{lo} - o_{cut} : end]$
12: $\hat{\mathbf{o}} \leftarrow [\mathbf{o}[0 : o_{lo}], \mathbf{o_{new}}]$
13: **if** $\mathbf{o_{new}} = Identity$ **then**
14: $\hat{L} \leftarrow \left[L_{00},\ 0;\ L_{10},\ \text{CHOL}(\Omega + L_{11}\, L_{11}^{\top})\right]$
15: $\hat{\mathbf{d}} \leftarrow \left[\mathbf{d}_0,\ \text{LSOLVE}(\hat{L}_{11}, \hat{\boldsymbol{\eta}}_1 - \hat{L}_{10}\, \mathbf{d}_0)\right]$
16: **else**
17: $\hat{\Lambda}_{ord_1} \leftarrow$ PERMUTE($\hat{\Lambda}, \hat{\mathbf{o}}$)$[o_{lo} : o_{hi}, :]$
18: $\hat{L} \leftarrow$ RESUMEDCHOL($L[0 : o_{lo}, :], \hat{\Lambda}_{ord_1}, o_{lo}$)
19: $\hat{\mathbf{d}} \leftarrow$ RESUMEDLSOLVE($[\hat{L}_{10}, \hat{L}_{11}], \hat{\boldsymbol{\eta}}_1, \mathbf{d}, o_{lo}$)
20: **end if**
21: **end if**

**Algorithm 2:** Incremental Block Cholesky Factorization.

distinct parts. The first part is keeping the $\Lambda$ matrix up to date. This can be done incrementally by adding $\Omega$, unless the linearization point changed. The change in the linearization point is stored in the $new_{LP}$ flag. The second part of the algorithm updates the $L$ factor along with the associated ordering and it is explained in detail in the next section. The third part of the algorithm is basically a simple Gauss-Newton nonlinear solver. An important point to note is that the nonlinear solver only needs to run if the residual grew after the last update. This is due to two assumptions; one is that the allowed number of iterations $maxIT$ is always sufficiently large to reach the local minima, and the other is that good initial priors are calculated. Without a loop closure, the norm of $\boldsymbol{\delta}$ would be close to zero and the system would not be updated. The first iteration uses updated $L$ factor, and the subsequent iterations use $\Lambda$ as it is much faster to be recalculated after the linearization point changed.

### *E. Incremental Block Cholesky Factorization*

In order to update the $L$ factor and the r.h.s. vector $\mathbf{b}$, it is possible to use (9) and (10), respectively. Without a proper ordering, $L$ will quickly become dense, slowing down the computation. $\Lambda$ can be reordered to reduce the fill-in. This has one major disadvantage that $L$ factor changes completely with the new ordering, impeding incremental factorization.

The solution is to only calculate the new ordering for parts of $L$ which are being affected by the update. In order to be able to calculate ordering incrementally, the updated $\hat{\Lambda}$ matrix is permuted with the ordering from the previous step, leading to $\hat{\Lambda}_p$ (see Fig. 3).

To delimit the area in $\hat{\Lambda}_p$ affected by the update, two indices are introduced. The first one, $o_{lo}$ is given by the minimum variable index after the ordering (line 6 of the Alg. 2). The second one, $o_{hi}$, is simply the size of the matrix. Let $\hat{\Lambda}_{p_{11}}$ be the lower right submatrix of $\hat{\Lambda}_p$ delimited by those indices.

Calculating the new ordering as AMD on $\hat{\Lambda}_{p_{11}}$ is not sufficient and also leads to massive fill-in. This is caused by the AMD library not having any information about the nonzero entries in $\hat{\Lambda}_{p_{10}} = \hat{\Lambda}_{p01}^{\top}$, which are also affected by this ordering (depicted by the blue blocks in Fig. 3).

A useful ordering can be calculated as AMD of full $\hat{\Lambda}_p$ with constraints applied to make sure that the ordering of the elements unaffected by the update is not disturbed. This is, however, a computationally expensive operation since the update is typically much smaller than $\hat{\Lambda}_p$, and a significant number of ordering constraints needs to be applied.

Fortunately, it is not necessary to calculate the ordering using the entire $\hat{\Lambda}_p$. It is possible to use a slightly expanded $\hat{\Lambda}_{p_{11}}$ (see Fig. 3) that satisfies the conditions of being square and not having any nonzero elements above or left from it ($\hat{\Lambda}_{p_{10}} = \hat{\Lambda}_{p_{01}}^{\top}$ are null). The ordering calculated on this submatrix is then combined with the original ordering (lines 11 and 12 in Alg. 2), yielding a similar result as constrained ordering on full $\hat{\Lambda}_p$ in much smaller time. The minimal size of the expanded $\hat{\Lambda}_{p_{11}}$ can be calculated in worst-case $O(n)$ time. First, a matrix wavefront is calculated. This is a vector containing the lowest positions of the nonzero elements per each column of $\hat{\Lambda}_p$. Only a part of this vector is used, the one between $o_{lo}$ and $o_{hi}$, and its minimum will give us the

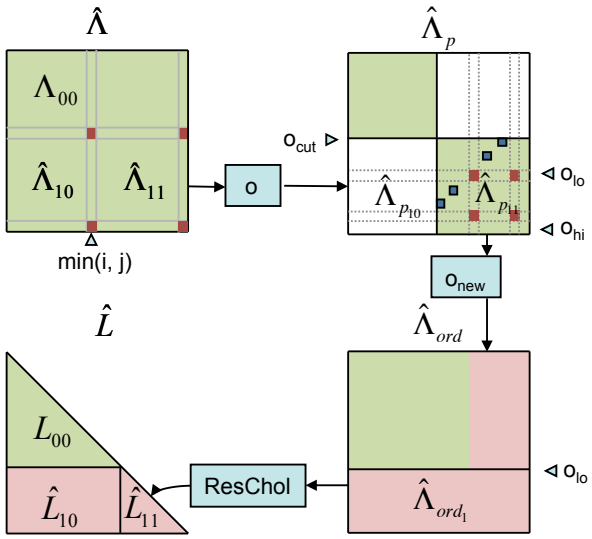


Fig. 3. Dataflow diagram of incremental block Cholesky factorization. Green parts of the matrices do not change, red parts represent the update and pink represents the parts that will change. The explanation is simplified to updates involving only two variables. Note that the green parts in $\hat{\Lambda}_p$ are unchanged respect to the previous step not respect to original $\hat{\Lambda}$.

index of the highest nonzero element, $o_{cut}$. This is done in Alg. 2 on line 9, and in the Fig. 3 it is depicted as the line, keeping the blue nonzero blocks out of $\hat{\Lambda}_{p_{10}}$. Extending $\hat{\Lambda}_{p_{11}}$ guarantees that AMD knows about all the nonzero elements that will cause fill-in, leading to a better ordering.

Once the new ordering is calculated, factorization can be performed. In the case that the ordering is identity, it is possible to only update $L_{11}$ and $d_1$ using (9, 10). Otherwise, the *resumed Cholesky* algorithm is employed. The row Cholesky is capable of calculating one row of the factor at a time, while only reading the values above it. This algorithm can be used to "resume" the factorization in the lower part of $L$ while only using the corresponding part of $\hat{\Lambda}_p$ and $L_{00}$ as inputs. The advantage of this algorithm is overall simplicity of the incremental updates to the factor, while also saving substantial time by avoiding recalculation of $L_{00}$, compared to the conventional approach.

Please note, that this is a simplified version of the algorithm, handling the type of updates where no new variables are introduced in the system. It is here where the fill-in is introduced and where the ordering is really needed.

## V. Experimental evaluation

This section evaluates both, the implementation of the incremental algorithm and of the incremental block Cholesky factorisation by comparing timing and the quality of the result with similar state of the art implementations. The evaluation was performed on five standard simulated datasets - *Manhattan* [15], *10k*, *City10k* [9], *CityTrees10k* [9] and *Sphere* [14] and four real datasets - *Intel* [8], *Killian Court* [2], *Victoria park* and *Parking Garage* [14]. Fig.4 shows the final solutions for all the tested datasets. All the tests were performed on an Intel Core $i5$ CPU $661$ with $8$ GB of RAM and running at $3.33$ GHz. This is a quad-core CPU without hyperthreading and with full SSE instruction set support. During the tests, the computer was not running any time-consuming processes in the background. Each test was run ten times and the average time was calculated in order to avoid measurement errors.

### A. Tested Implementations

We compared the proposed incremental algorithm and its implementation with state of the art implementations such as g2o [14], iSAM [10] and the gtsam implementation of the iSAM2 algorithm [11, 12]. For SPA the svn revision 39478 of ROS (http://www.ros.org/) was used; for g2o, we tested the version 91A858D available at https://github.com/RainerKuemmerle/g2o.[1] For iSAM we used the version 1.6 from https://svn.csail.mit.edu/isam and for gtsam we used the version 2.3 from https://collab.cc.gatech.edu/borg/gtsam.

SPA and g2o are both based on similar sparse block matrix scheme which is maintained until the matrix factorization is performed. At this point, they switch to CCS format to be able to use CSparse or CHOLMOD to perform the factorization. This is a time consuming process which is avoided in our approach. While SPA implementation is optimized for the specific 2D SLAM problem, g2o is general, allowing any type of SLAM and BA.

iSAM and iSAM2 are based on completely different algorithms. The one used in iSAM is very similar to our algorithm but it requires periodic batch steps to reduce the fill-in. The algorithm used in iSAM2 is based on the Bayes tree data structure and the factorization is done through elimination on factor graphs. One important characteristic is that it allows incremental reordering and fluid relinearization. In this direction, our algorithm allows similar incremental reordering but changes the entire linearization point when needed. In order to test the iSAM2 we used the incremental test example from the library and extended it to work with landmark-based and 3D SLAM datasets.

The proposed block Cholesky (BC) factorization is part of a new nonlinear least squares open-source library available for download at http://sourceforge.net/projects/slam-plus-plus/. The main characteristic of this new library is its ability to manipulate block matrices and to produce efficient incremental solutions. In this paper we test the BC factorization on both, an algorithm that operates only on the information matrix $\Lambda$ performing batch updates every step (denoted $Inc\_\Lambda$) and an incremental algorithm, which maintains the matrix factorization $L$ up to date (denoted $Inc\_L$). The latter corresponds to the algorithm in 1. The new library offers the possibility to switch between the native BC factorization and the Cholesky factorization form CSparse (CS) and CHOLMOD (CM).

### B. Performance and Accuracy

Table I shows the execution times and accuracy of the above described implementations evaluated on the datasets in

[1]We thank the authors for providing the link and support

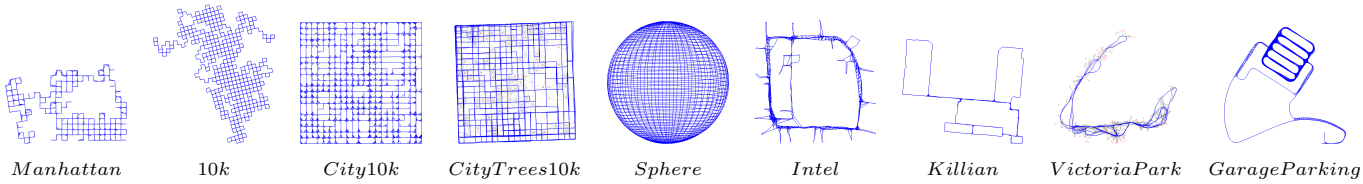


Fig. 4. The datasets used in our evaluations. "Sphere"and "Garrage Parking" datasets are 3D pose graphs.

| | Manhattan | 10K | City10k | CityTrees10k | Sphere | Intel | Killian | Victoria Park | Parking Garage |
|---|---|---|---|---|---|---|---|---|---|
| $SPA$ | 24.161 | 518.339 | 309.562 | $N/A$ | $N/A$ | 1.486 | 5.669 | $N/A$ | $N/A$ |
| $g2o$ | 22.514 | 500.374 | 302.495 | 175.124 | 145.486 | 1.298 | 5.019 | 81.194 | 20.372 |
| $iSAM$ ($b100$) | 4.829 | 279.926 | 77.572 | 22.926 | 36.220 | 1.287 | 4.213 | 11.921 | 52.2167 |
| $iSAM2$ | 4.932 | 91.738 | 60.978 | 32.687 | 31.274 | 0.618 | 1.196 | 16.349 | 3.658 |
| $Inc\_\Lambda - CS$ | 8.603 | 287.702 | 202.839 | 19.531 | 216.487 | 0.651 | 1.705 | 23.162 | 17.317 |
| $Inc\_\Lambda - CM$ | 10.725 | 236.276 | 181.139 | 24.478 | 71.487 | 0.786 | 2.100 | 28.264 | 23.929 |
| $Inc\_\Lambda - BC$ | 7.209 | 242.209 | 188.849 | **17.566** | 78.375 | 0.508 | 1.242 | 18.707 | 11.342 |
| $Inc\_L - BC$ | **3.046** | **79.651** | **53.951** | 19.308 | **9.865** | **0.353** | **1.045** | **11.202** | **3.410** |
| $\chi^2$ $iSAM2$ | 6205.92 | 171600 | 31951.6 | 794.868 | 775.28 | 559.07 | 0.00008 | 370.14 | 1.2635 |
| $\chi^2$ $Inc\_L - BC$ | 6119.83 | 171919 | 31931.4 | 12062.6 | 727.72 | 558.83 | 0.00005 | 144.91 | 1.3106 |

TABLE I
PERFORMANCE AND ACCURACY TESTS ON MULTIPLE DATASETS.

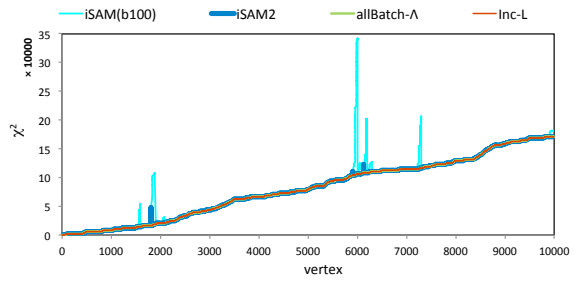


Fig. 5. Quality of the estimations measured on $10k$ dataset.

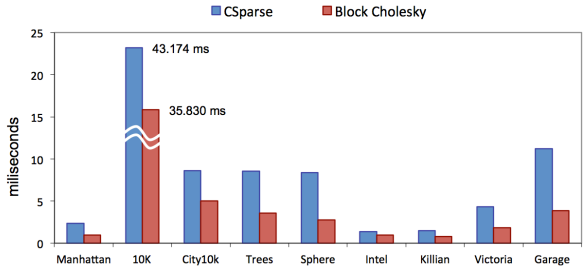


Fig. 6. Cholesky factorization benchmark.

Fig.4. For every test we evaluate both, building the system and computing the solution. The first one is necessary because changing the matrix numerically and structurally is different for each implementation and this makes significant difference in an incremental approach.

The proposed incremental algorithm is different from the one employed in SPA and g2o, where the batch solve is done every $n$ new variables added to the system and no solutions are available in between. Therefore, the time comparison with these implementation is orientative. The comparison stays only for $n = 1$, where the solution is available every step and our $Inc\_\Lambda$ solver. iSAM, iSAM2 and $Inc\_L$ provide solution every step. The main difference is that iSAM requires the periodic batch solves, the default setting of $n = 100$ is used in the comparison. But keeping the same linearization point for too long deteriorate the estimation. This can be seen by plotting the the sum of squared errors ($\chi^2$ in Fig. 5). Spikes appear when performing periodic batch solve due to the fact that the error increases between the batch steps and drops afterwards. Observe that $Inc-L$ (in red in Fig. 5) and iSAM2 nicely follow the $Inc-\Lambda$ (in green in Fig. 5), which represent the most accurate solution one can get solving the nonlinear problem.

Our implementation reaches the best times for the best accuracy on all evaluated datasets and this is shown in bold in table I. Except for the *CityTrees10k* dataset, the execution of the $Inc\_L$ outperforms all the implementations. This particular result is given by the dense structure of the problem. In this case, reordering every step is slightly more advantageous than incremental ordering. The closest time to $Inc\_L$ is reached by the iSAM2. The difference between iSAM2 and $Inc\_L$ is that iSAM2 changes only the affected blocks of the $L$ factor and relinearizes only affected variables, while $Inc\_L$ changes parts of the $L$ factor and relinearizes all the affected variables when needed. iSAM2 was run with a defaull reliniarization threshold set to 10. This leads to slightly better accuracy of the estimation provided by $Inc\_L$ (see the last two rows of the table I) and favours iSAM2 in the execution time comparison.

The proposed Cholesky factorization algorithm was tested on full system matrices of the same datasets used in the incremental algorithm evaluation. The results are shown in Fig. 6. Our block Cholesky implementation (BC) is always

faster than the CSparse (v3.0.2) and CHOLMOD (v2.1.2). Also note that the speedup grows with the block size, for $6 \times 6$ blocks it is more than double. The quality of the factorization is also good, the worst norm of difference between block Cholesky and CSparse was $2.6016 \cdot 10^{-13}$ and occurred on the City10k dataset.

## VI. Conclusion and Future Work

A new incremental NLS algorithm with applications to robotics was proposed in this paper. We targeted problems such as SLAM, which have a particular block structure, where the size of the blocks corresponds to the number of degrees of freedom of the variables. This enabled several optimizations which made our implementation faster than the state-of-the-art implementations, while achieving very good precision. This was demonstrated through the comparison with the existing implementations on several standard datasets.

Recently, many efforts have been made to develop both, efficient incremental algorithms and implementations. This paper complements the recent advances by introducing new incremental ordering scheme which allows to incrementally update the factorized form of the linearized system while maintaining a reduced fill-in. The incremental updates are done using a resumed block Cholesky factorization only on the parts affected by the new information. The block Cholesky factorization itself proved to be more efficient than the current implementations of elementwise Cholesky factorizations while the precision is equally high.

Several further algorithmic improvements can be made. It is possible to update only the affected blocks in $L$, instead of a whole part of the factor, and to change only the affected variables in a similar way iSAM2 does. Also, similarly to how the matrix factorization and the associated ordering are updated incrementally, other parts of the algorithm can be optimized, such as the calculation of the AMD ordering heuristic.

The data structure was designed with hardware acceleration in mind. This is very important for large scale problems, which can run faster on a wide range of accelerators, from multicore CPUs to clusters of GPUs.

## Acknowledgments

The research leading to these results has received funding from the EU $7^{th}$ FP, grants 316564-IMPART and 247772-SRS, Artemis JU grant 100233-R3-COP, and the IT4Innovations Centre of Excellence, grant n. CZ.1.05/1.1.00/02.0070, supported by Operational Programme Research and Development for Innovations funded by Structural Funds of the European Union and the state budget of the Czech Republic.

## References

[1] P. Amestoy, T. A. Davis, and I. S. Duff. Amd, an approximate minimum degree ordering algorithm). *ACM Transactions on Mathematical Software*, 30(3):381–388, 2004.

[2] M.C. Bosse, P.M. Newman, J.J. Leonard, and S. Teller. Simultaneous localization and map building in large-scale cyclic environments using the Atlas framework. *Intl. J. of Robotics Research*, 23(12):1113–1139, Dec 2004.

[3] Tim Davis. Csparse. http://www.cise.ufl.edu/research/sparse/CSparse/, 2006.

[4] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, 2006. ISBN 0898716136.

[5] Timothy A. Davis and William W. Hager. Modifying a sparse cholesky factorization, 1997.

[6] F. Dellaert and M. Kaess. Square Root SAM: Simultaneous localization and mapping via square root information smoothing. *Intl. J. of Robotics Research*, 25(12):1181–1203, Dec 2006.

[7] G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard. A tree parameterization for efficiently computing maximum likelihood maps using gradient descent. In *Robotics: Science and Systems (RSS)*, Jun 2007.

[8] A. Howard and N. Roy. The robotics data set repository (Radish), 2003. URL http://radish.sourceforge.net/.

[9] M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Fast incremental smoothing and mapping with efficient data association. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 1670–1677, Rome, Italy, April 2007. ISBN 1-4244-0601-3.

[10] M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Incremental smoothing and mapping. *IEEE Trans. Robotics*, 24(6):1365–1378, Dec 2008.

[11] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. Leonard, and F. Dellaert. iSAM2: Incremental smoothing and mapping with fluid relinearization and incremental variable reordering. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.

[12] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert. iSAM2: Incremental smoothing and mapping using the Bayes tree. *Intl. J. of Robotics Research*, 31:217–236, February 2012.

[13] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, and R. Vincent. Efficient sparse pose adjustment for 2d mapping. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, Taipei, Taiwan, October 2010.

[14] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A general framework for graph optimization. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, Shanghai, China, May 2011.

[15] Edwin Olson. *Robust and Efficient Robot Mapping*. PhD thesis, Massachusetts Institute of Technology, 2008.