Incremental Cholesky Factorization for Least Squares Problems in Robotics ⋆

Lukas Polok, Marek Solony, Viorela Ila, Pavel Smrz and Pavel Zemcik *

* *Brno University of Technology, Faculty of Information Technology. Bozetechova 2, 612 66 Brno, Czech Republic*
({ipolok,isolony,ila,smrz,zemcik}@fit.vutbr.cz)

Abstract: Online applications in robotics, computer vision, and computer graphics rely on efficiently solving the associated nolinear systems every step. Iteratively solving the non-linear system every step becomes very expensive if the size of the problem grows. This can be mitigated by incrementally updating the linear system and changing the linearization point only if needed. This paper proposes an incremental solution that adapts to the size of the updates while keeping the error of the estimation low. The implementation also differs form the existing ones in the way it exploits the block structure of such problems and offers efficient solutions to manipulate block matrices within incremental nonlinear solvers. In this work, in particular, we focus our effort on testing the method on simultaneous localization and mapping (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 tested datasets.

Keywords: Robotics, Least squares problems, SLAM, Incremental solvers

1. INTRODUCTION

Many problems in robotics, computer vision and computer graphics can be formulated as nonlinear least square optimization. The goal is to find the optimal configuration of the variables that maximally satisfy the set of nonlinear constraints. For instance in robotics, SLAM finds the optimal configuration of the robot positions and/or landmarks in the environment given a set of measurements from the robot sensors. In computer vision, structure from motion (SfM) and bundle adjustment (BA) problems are mathematically equivalent to SLAM, where the variable set includes all camera poses and 3D object points, with some slight differences on the types of constraints, in computer vision we emphasise on uncalibrated setups and associated self-calibration methods.

Finding the optimal configuration, often called the maximum likelihood, is obtained by solving a sequence of least-squares minimization problems. In practice, the initial problem is nonlinear and it is usualy addressed by repeatedly solving a sequence of linear systems in an iterative Gauss-Newton (GN) or Levenberg-Marquardt (LM) nonlinear solver or recently used in SLAM, Powell's Dog-Leg trust-region method Rosen et al. (2012). The linearized system can be solved either using direct methods, such as matrix factorization or iterative methods, such as conjugate gradients. Iterative methods are more efficient from the storage (memory) point of view, since they only require access to the gradient, but they can suffer from poor convergence, slowing down the execution. Direct methods, on the other hand, produce more accurate solutions and avoid convergence difficulties but they typically require a lot of storage.

The challenge appears in online applications, where the state changes every step. In an online SLAM application, for example, every step the state is *incremented* with a new robot position and/or a new landmark and it is *updated* with the corresponding measurements. For very large problems, updating and solving the nonlinear system in every step can become very expensive. Every iteration of the nonlinear solver involve building a new linear system using the current linearization point. This can be alleviated by changing the linearization point only when the error increases. This means solving the nonlinear systems only when needed and in between providing an approximate estimate of the solution computed at the last linearization point.

The solution is to incrementally update the linear system in the already factorized form and perform one backsubstitution to compute the solution. This idea was introduced in Kaess et al. (2008) where the factor R, obtained applying QR factorization of the linear system matrix, is updated every step using Givens rotations. This method becomes advantageous when the *batch steps* are performed periodicaly. The batch steps involve changing the linearization points by solving the nonlinear system. They 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 factor R increases with the incremental updates slowing down the backsubstitutions. Therefore, the system proposed in Kaess et al. (2008) performs such

⋆ The research leading to these results has received funding from the European Union, 7th Framework Programme, 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.

periodic updates, typically every 10 or every 100 steps to obtain efficient incremental solutions.

The new method introduced in this paper has the advantage that it adapts to the size of the updates and performs batch steps only when needed while still keeping the option to set the frequency of the batch steps. It is based on several optimizations of the incremental algorithm and its implementation a) selects between three types of updates, depending on the size of the the update and the error b) uses double-constrained ordering by blocks c) performs backsubstitution by blocks d) uses efficient block-matrix scheme for storage and arithmetic operations. These optimizations allow for very fast online execution of the algorithm and provide very accurate solutions every step.

2. RELATED WORK

Several successful implementations of nonlinear least squares optimization techniques for SLAM already exist and have been used in robotic applications. In general, they are based on similar algorithmic framework, repeatedly applying Cholesky or QR factorizations in an iterative Gauss-Newton or Levenberg-Marquardt nonlinear solver. g2o Kümmerle et al. (2011) is an easy to use, open-source implementation which has been proven to be very fast in batch mode. It exploits the sparse connectivity and operates on the block-structure of the underlying graph problem. A similar scheme was initially implemented in SSBA Konolige (2010) and SPA K. Konolige and R.Vincent (2010) and it is based on block-oriented sparse matrix manipulation. Using blocks is a natural way to optimize the storage, nevertheless, taking care about the layout of the individual blocks in the memory is very important, otherwise the overhead of handling the blocks can easily outweigh the advantage of cache efficiency. Our implementation takes care about this aspect, providing increased efficiency.

However, in SLAM the state changes every step when new observations need to be integrated into the system. For very large problems, updating and solving every step can become very expensive. Incremental smoothing and mapping (iSAM) allows efficiently solving a nonlinear optimization problem in every step Kaess et al. (2008). The implementation incrementally updates the R factor obtained from the QR factorization and performs back-substitution to find the solution. The sparsity of the R factor is ensured by periodic reordering. Recently, the Bayes tree data-stucture Kaess et al. (2010, 2011, 2012) was introduced to enable a better understanding of the link between sparse matrix factorization and inference in graphical models. The Bayes tree was used to obtain iSAM2 Kaess et al. (2011, 2012), which achieves high efficiency through incremental variable reordering and fluid relinearization, eliminating the need for periodic batch steps. When compared to the existing methods, iSAM2 performance finds a good balance between efficiency and accuracy. But still the complexity of maintaining the Bayes tree data structure can introduce several overheads.

The solution proposed in this paper provides accurate solutions every step and has an increased efficiency through above mentioned optimizations. The paper is structured as follows. The next section succinctly formalizes SLAM as a nonlinear least squares problem. Incremental updates are described in Section 4. Then, in Section 5 we introduce the new algorithm and the characteristics of our new implementation. In the experimental evaluation in Section 6 we show the increased efficiency of our proposed scheme over the existing implementations. Conclusions and future work are given in Section 7.

3. SLAM AS A NONLINEAR LEAST SQUARES PROBLEM

In robotics, SLAM is often formulated as a nonlinear least squares problem Dellaert and Kaess (2006), which estimates a set of variables $\boldsymbol{\theta} = [\theta_1 \dots \theta_n]$ containing the robot trajectory and the position of landmarks in the environment, given a set of measurement constraints $\mathbf{z}$ between those variables. The constraints come from control inputs and measurements (odometric, vision, laser, etc.). The joint probability distribution can be written as:

$$P(\boldsymbol{\theta}, \mathbf{z}) \propto P(\theta_0) \prod_{z}^{n} P(z_k \mid \theta_{i_k}, \theta_{j_k}), \tag{1}$$

where $P(\theta_0)$ is the prior and z_k are the constraints between the variables θ_{i_k} and θ_{j_k}. The goal is to obtain the maximum likelihood estimate (MLE) for all variables in $\boldsymbol{\theta}$, given the measurements in $\mathbf{z}$:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\ P(\boldsymbol{\theta} \mid \mathbf{z}) = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\ \{-\log(P(\boldsymbol{\theta} \mid \mathbf{z})\}\ . \tag{2}$$

In SLAM, those constraints involve rotations and are nonlinear. For every measurement z_k, we assume Gaussian distributions:

$$z_k = h_k(\theta_{i_k}, \theta_{j_k}) - v_k\ , \tag{3}$$

$$P(z_k \mid \theta_{i_k}, \theta_{j_k}) \propto \exp\left(-\frac{1}{2} \parallel h_k(\theta_{i_k}, \theta_{j_k}) - z_k \parallel^2_{\Sigma_k}\right), \tag{4}$$

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 Σ_k. Finding the MLE from (2) is done by solving the following nonlinear least squares problem:

$$\boldsymbol{\theta}^* = \operatorname{argmin} \left\{ \sum_{k=1}^{m} \|h_k(\theta_{i_k}, \theta_{j_k}) - z_k\|^2_{\Sigma_k} \right\}\ , \tag{5}$$

where we minimize the sum of squared residual of the type:

$$r_k = h_k(\theta_{i_k}, \theta_{j_k}) - z_k\ . \tag{6}$$

Gathering all residuals in $\mathbf{r}(\boldsymbol{\theta}) = [r_1, \dots, r_m]^\top$ and the measurement noise in $\boldsymbol{\Sigma} = diag([\Sigma_1, \dots, \Sigma_m])$, the sum in (5) can be written in the vectorial form and expressed in terms of 2-norm:

$$\|\mathbf{r}(\boldsymbol{\theta})\|^2_{\boldsymbol{\Sigma}} = \mathbf{r}^\top(\boldsymbol{\theta})\, \boldsymbol{\Sigma}^{-1} \mathbf{r}(\boldsymbol{\theta}) = \left\| \boldsymbol{\Sigma}^{-\top \backslash 2} \mathbf{r}(\boldsymbol{\theta}) \right\|^2\ . \tag{7}$$

Methods such as Gauss-Newton or Levenberg-Marquardt are often used to solve the nonlinear problem in (5) and this is usualy addressed by iteratively solving the sequence of linear systems. Those are obtained based on a series of linear approximations of the nonlinear functions $\mathbf{r}$ around the current linearization point $\boldsymbol{\theta}^0$:

$$\tilde{\mathbf{r}}(\boldsymbol{\theta}^0) = \mathbf{r}(\boldsymbol{\theta}^0) + J(\boldsymbol{\theta}^0)(\boldsymbol{\theta} - \boldsymbol{\theta}^0)\ , \tag{8}$$

where J is the Jacobian matrix which gathers the derivative of the components of $\mathbf{r}(\boldsymbol{\theta}^0)$. The linearized problem to solve becomes:

$$\boldsymbol{\delta}^* = \operatorname{argmin} \|A\, \boldsymbol{\delta} - \mathbf{b}\|^2\ , \tag{9}$$

where the $A = \mathbf{\Sigma}^{-\top\backslash 2} J$ is the system matrix, $\mathbf{b} = -\mathbf{\Sigma}^{-\top\backslash 2}\mathbf{r}$ the right hand side (r.h.s.) and $\boldsymbol{\delta} = (\boldsymbol{\theta} - \boldsymbol{\theta}^0)$ the correction to be calculated. This is a standard least squares problem in $\boldsymbol{\delta}$. For SLAM problems, the matrix A is in general sparse, but it can become very large when the robot performs long trajectories. The normalized system has the advantage of remaining of the size of the state even if the number of measurements increases:

$$\boldsymbol{\delta}^* = \operatorname{argmin} \|\Lambda\boldsymbol{\delta} - \boldsymbol{\eta}\|^2 \ , \tag{10}$$

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

3.1 Linear Solver

The linearized version of the problem introduced above can be efficiently solved using sparse direct optimization methods based on factorizing the system matrices A or Λ followed by backsubstitution.

QR factorization can be applied directly to the matrix A in (9), yielding $A = Q\, R$. The solution $\boldsymbol{\delta}$ can be directly obtained by backsubstitution in $R\boldsymbol{\delta} = \mathbf{d}$ where $\mathbf{d} = R^{-\top}\, A^\top\, \mathbf{b}$. Note, that Q is not explicitly formed; instead $\mathbf{b}$ is modified during factorization to obtain $\mathbf{d}$.

Cholesky factorization yields $\Lambda = L\, L^\top$, where $L = R^\top$ is the Cholesky factor, and a forward and back substitutions on $L\mathbf{d} = A^\top \mathbf{b}$ and $L^\top\, \boldsymbol{\delta} = \mathbf{d}$ first recovers $\mathbf{d}$, then the actual solution $\boldsymbol{\delta}$. In general, QR factorization has better numerical properties but Cholesky factorization performs faster.

3.2 Incremental SLAM

Online robotic applications require fast and accurate methods for the estimation of the current position of the robot and of the map. In an online application, the state is *incremented* with a new robot position and/or a new landmark every step and it is *updated* with the corresponding measurements.

The system in (9) can be incrementally built by appending the matrix A with new columns corresponding to each new variable (pose/landmark) and new rows corresponding to each measurement. For each new measurement, the new block row is sparse and the only nonzero elements correspond to the Jacobians of the new residual.

For the normalized system in (10), the size of the matrix increments in number of rows and columns with the size of each new variable and it is updated by adding the new information to Λ and η. For simplicity of the notations, in the following formulations, we drop the k subindices since it will always refer to the last measurement, instead, the system matrices are split in parts that change (e.g. Λ_{22}) and parts that remains unchanged (e.g. Λ_{11} and Λ_{21}). To match with the formulation in section 3, we keep the subindex k for the kth residual and its corresponding covariance. With that, the update step can be written as:

$$\tilde{\Lambda} = \begin{pmatrix} \Lambda_{11} & \Lambda_{21}^\top \\ \Lambda_{21} & \Lambda_{22} + \Omega \end{pmatrix} \ ; \ \tilde{\boldsymbol{\eta}} = \begin{bmatrix} \boldsymbol{\eta}_1 \\ \boldsymbol{\eta}_2 + \boldsymbol{\omega} \end{bmatrix} \ , \tag{11}$$

where $\Omega = H^\top \Sigma_k^{-1} H$, $\boldsymbol{\omega} = -H\Sigma_k^{-1\backslash 2} r_k$ and H is:

$$H = \begin{bmatrix} J_i^j \ \ldots 0 \ldots \ J_j^i \end{bmatrix} \ , \tag{12}$$

J_i^j and J_j^i are the derivatives of the $r(\cdot)$ function with respect to θ_i and θ_j. The sparsity and the size of the Ω matrix are important for the incremental updates of the system:

$$\Omega = \begin{pmatrix} J_i^j\, \Sigma_k^{-1}\, J_i^{j\top} & \ldots 0 \ldots & J_i^j\, \Sigma_k^{-1}\, J_j^{i\top} \\ \vdots & & \vdots \\ 0 & \ldots & 0 \\ \vdots & & \vdots \\ J_j^i\, \Sigma_k^{-1}\, J_i^{j\top} & \ldots 0 \ldots & J_j^i\, \Sigma_k^{-1}\, J_j^{i\top} \end{pmatrix} . \tag{13}$$

For very large problems, updating every step become very expensive. Therefore, this paper proposes an efficient method to update directly the matrix factorization only when needed and provide good estimation every step.

4. INCREMENTALLY UPDATING THE CHOLESKY FACTOR

In this section we show how to directly update the Cholesky factor $L = R^\top$ in order to avoid unnecessary and expensive matrix factorizations every step. Observe that in (11) only a part of the information matrix and the information vector is changed in the update process and the same happens with the lower trinagular factor L. The updated $\tilde{L}$ factor and the corresponding r.h.s. $\tilde{\mathbf{d}}$ can be written as:

$$\tilde{L} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & \tilde{L}_{22} \end{pmatrix} \ ; \ \tilde{\mathbf{d}} = \begin{bmatrix} \mathbf{d_1} \\ \tilde{\mathbf{d_2}} \end{bmatrix} \ . \tag{14}$$

From $\tilde{\Lambda} = \tilde{L}\, \tilde{L}^\top$, (11) and (14) it derives:

$$\Lambda_{22} + \Omega = L_{21}\, L_{21}^\top + \tilde{L}_{22}\, \tilde{L}_{22}^\top \ , \tag{15}$$

and the part of the $\tilde{L}$ factor that changes after the update can be computed by applying Cholesky decomposition to a reduced size matrix:

$$\tilde{L}_{22} = chol(\Lambda_{22} + \Omega - L_{21}\, L_{21}^\top) \ , \tag{16}$$

$$\tilde{L}_{22} = chol(\tilde{\Lambda}_{22} - L_{21}\, L_{21}^\top) \ , \tag{17}$$

$$\tilde{L}_{22} = chol(L_{22}\, L_{22}^\top + \Omega) \ . \tag{18}$$

Further in this paper we will refer to (17) as Lambda-updates because it uses parts of the $\tilde{\Lambda}$ to update L and to (18) as Omega-updates because it directly uses Ω to update L.

The part of the r.h.s. affected by the new measurement can also be easily updated:

$$\tilde{\mathbf{d_2}} = \tilde{L}_{22} \backslash (\boldsymbol{\eta}_2 + \boldsymbol{\omega} - L_{21}\, \mathbf{d_1}) \tag{19}$$

$$\tilde{\mathbf{d_2}} = \tilde{L}_{22} \backslash (\tilde{\boldsymbol{\eta}}_2 - L_{21}\, \mathbf{d_1}) \tag{20}$$

where the $\backslash$ is the matrix left division operator. After obtaining $\tilde{L}$ and $\tilde{\mathbf{d}}$, backsubstitution is performed to find the solution of the linear system $\tilde{L}^\top\, \boldsymbol{\delta} = \tilde{\mathbf{d}}$.

In mobile robotic applications, odometric measurements have higher rate but they link only consecutive robot

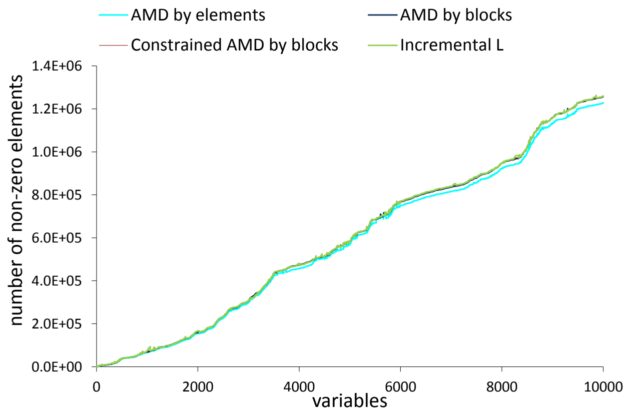

Fig. 1. The comparison in terms of nonzero elements of several ordering heuristics and the actual number of non-zero elements in Incremental L.

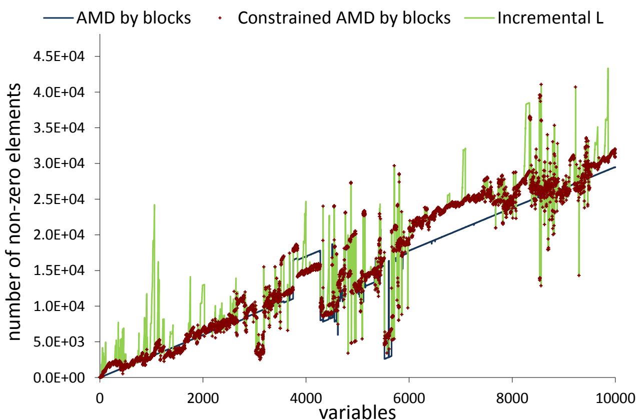

Fig. 2. The fill-in relative to the best heuristic in Fig.1, which is AMD by elements.

poses, which translate in small Ω sizes and fast updates. Loop-closure involve links between variables far apart in the system, and the updates can be slower. Next section proposes solutions to those problems.

5. IMPLEMENTATION DETAILS

Online applications such as SLAM, require extremely fast methods for building, updating and solving the sequence of linearized systems. In this section, we introduce several optimizations towards high performance SLAM based on incremental updates of the factored representation.

5.1 Adaptive updates

The proposed methodology adapts to the most favourable incremental update scheme, depending on the size of the updates. It considers three ways to update the system: 1) Omega-updates, 2) Lambda-updates and 3) updating the entire L, and applies heuristics to select the best strategy. Omega-updates in (18) are fast for small-size Ω because they involve the multiplication of small matrices $L_{22}\, L_{22}^{\top}$ but with large fill-in. Therefore, this is not suitable when Ω is obtained from measurements that are far apart (e.g. loop-closure). In this case Lambda-updates in (17) are faster since they involve the multiplication of very sparse matrices $L_{21}\, L_{21}^{\top}$.

Updating very large loops becomes expensive due to bookkeeping. When loop length approaches the number of variables in the system, recalculating L by applying Cholesky decomposition to the Λ matrix becomes more efficient. Full factorization can be slow, however, due to the fact that the ordering heuristics are applied to the entire Λ, it considerably reduces the fill-in of the factor L and speeds up the backsubstitution.

5.2 Efficient ordering strategies

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 Λ, 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 P. Amestoy and Duff (2004) 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 Davis (2006).

In an incremental SLAM process, the new variable, either the next observed landmark or the next robot pose, is always linked to the current 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. This especially helps when updating using odometric constraints between consecutive poses. For landmark SLAM, one landmark is often observed from several poses. Without an additional constraint, a recently observed landmark can be ordered anywhere in the matrix, possibly causing large-size updates. To alleviate this problem, our implementation constraints recently observed landmarks to immediately precede the last pose. Figures 1 and 2 show that the used ordering restrictions barely affect the fill-in. 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 in 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.

5.3 Block matrix scheme

SLAM involves 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 Davis (2006) or CHOLMOD Davis and Hager (1997) 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 modification, at the cost of slight memory overhead, and slightly worse arithmetic efficiency. Since block sizes in SLAM problems are

known in advance, the individual blocks in a sparse block matrix can be processed using vectorized SSE instructions and the performance is increased. Our implementation combines the advantages of *block-wise* schemes convenient in both, numeric and structural matrix modification and *element-wise*, which allows efficient arithmetic operation on sparse matrices.

On the other hand, some operations are faster when performed element-wise and in a dense fashion. For example applying dense Cholesky on fixed-size matrices is faster than sparse Cholesky, up to certain size where faster SSE implementation gets beaten by the fact that it operates mostly on zeroes when L is very sparse. Therefore, dense Cholesky is applied for up to $5\,blocks \times 5\,blocks$ matrices.

5.4 Other optimizations

Backsubstitution has proven to be more advantageous when performed block-wise. Since the r.h.s. vector is dense, it is possible to accelerate backsubstitution using SSE, similar to the other operations. Also, in the context of our implementation, having backsubstitution performed block-wise avoids converting the block-wise L factor to a sparse element-wise matrix.

For simplicity, the formulation introduced in Section 4 is done using the lower-triangular factor L. The implementation uses the upper-triangular $R = L^\top$ for several reasons. The most important fact is that the CHOLMOD and CSparse libraries only use upper-triangular part of a matrix to calculate Cholesky factorization. That means only the upper-triangular part of Λ needs to be calculated and stored. In order to be able to perform the partial updates of the factor using this upper triangular Λ, the factor also needs to be upper triangular.

Some of the operations can benefit from keeping Λ up to date at all times and our implementation allows for efficient storage of sparse block matrices. Furthermore, incrementally updating Λ is virtually free using our block matrix schemes, as updates to Λ are additive. Therefore, the implementation keeps both, Λ and R.

5.5 Incremental Algorithm

Our approach is described by pseudocode in Alg. 1. It can be seen as having three distinct parts. The first part is keeping the Λ matrix up to date. This can be done incrementally by adding Ω, unless the linearization point changed. The change in the linearization point is stored in the $have_\Lambda$ flag.

The second part of the algorithm updates the L factor. The algorithm employs a simple heuristic to decide which update method is the best. In case of large updates, invalidating a substantial portion of L, or if the linearization point changed, L is recalculated from Λ. This step involves calculating a suitable ordering using the constrained AMD algorithm. On the other hand, if L is up to date and the size of the update is relatively small, it is faster to update L using either (18), which is faster for very short updates, or using (17). The r.h.s. vector d is updated in a similar manner.

$\boldsymbol{\theta}_c \leftarrow$ INCREMENT-L($\boldsymbol{\theta}, \mathbf{r}, \boldsymbol{\Sigma}_z, L, \mathbf{d}, \Lambda, \boldsymbol{\eta}, have_L, have_\Lambda, maxIT, tol$)

1: $(\Omega, \boldsymbol{\omega}) \leftarrow$ COMPUTEOMEGA($(\theta_{ik}, \theta_{jk}), r_k, \Sigma_k$)
2: **if** $\neg have_\Lambda$ **then**
3: $(\Lambda, \boldsymbol{\eta}) \leftarrow$ LINEARSYSTEM($\boldsymbol{\theta}, \mathbf{r}$)
4: $have_\Lambda \leftarrow$ TRUE
5: **else**
6: $(\Lambda, \boldsymbol{\eta}) \leftarrow$ UPDATELINEARSYSTEM($\Lambda, \eta, \Omega, \boldsymbol{\omega}$)
7: **end if**
8: $loopSize \leftarrow$ COLUMNS(Ω)
9: **if** $\neg have_L \,\|\, loopSize > bigLoopThresh$ **then**
10: $L \leftarrow$ CHOL(Λ)
11: $\mathbf{d} \leftarrow$ LSOLVE($L, \boldsymbol{\eta}$)
12: $have_L \leftarrow$ TRUE
13: **else**
14: **if** $loopSize < smallLoopThresh$ **then**
15: $L \leftarrow \left[L_{11},\, 0;\, L_{21},\, \text{CHOL}(\Omega + L_{22}\, L_{22}^\top)\right]$
16: **else**
17: $L \leftarrow \left[L_{11},\, 0;\, L_{21}, \text{CHOL}(\Lambda_{22} - L_{21}\, L_{21}^\top)\right]$
18: **end if**
19: $\mathbf{d} \leftarrow [\mathbf{d}_1;\, \text{LSOLVE}(L_{22}, \boldsymbol{\eta}_2 - L_{21}\,\mathbf{d}_1)]$
20: **end if**
21: **if** $maxIT \leq 0 \,\|\, \neg\, hadLoop$ **then**
22: **exit**
23: **end if**
24: $it = 0$
25: **while** $it < maxIT$ **do**
26: **if** $it > 0$ **then**
27: $(\Lambda, \boldsymbol{\eta}) \leftarrow$ LINEARSYSTEM($\boldsymbol{\theta}, \mathbf{r}$)
28: $\boldsymbol{\delta} \leftarrow$ CHOLSOLVE($\Lambda, \boldsymbol{\eta}$)
29: $have_\Lambda \leftarrow$ TRUE
30: **else**
31: $\boldsymbol{\delta} \leftarrow$ LSOLVE($L, \mathbf{d}$)
32: **end if**
33: **if** $norm(\boldsymbol{\delta}) \geq tol$ **then**
34: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} \oplus \boldsymbol{\delta}$
35: $have_\Lambda \leftarrow$ FALSE
36: $have_L \leftarrow$ FALSE
37: **else**
38: **exit**
39: **end if**
40: $it++$
41: **end while**

Algorithm 1: Incremental-L algorithm.

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 δ would be close to zero and the system would not be updated anyway. The first iteration uses updated L factor, and the subsequent iterations use Λ as it is much faster to be recalculated after the linearization point changed.

6. EXPERIMENTAL RESULTS

In order to evaluate the proposed incremental algorithm and its implementation this section compares timing and sum of squared errors with similar state of the art implementations such as iSAM Kaess et al. (2008), g2o Kümmerle et al. (2011), and SPA K. Konolige and R.Vincent (2010) (a 2D SLAM variant of SSPA). These implementations are easy to use on standard datasets.

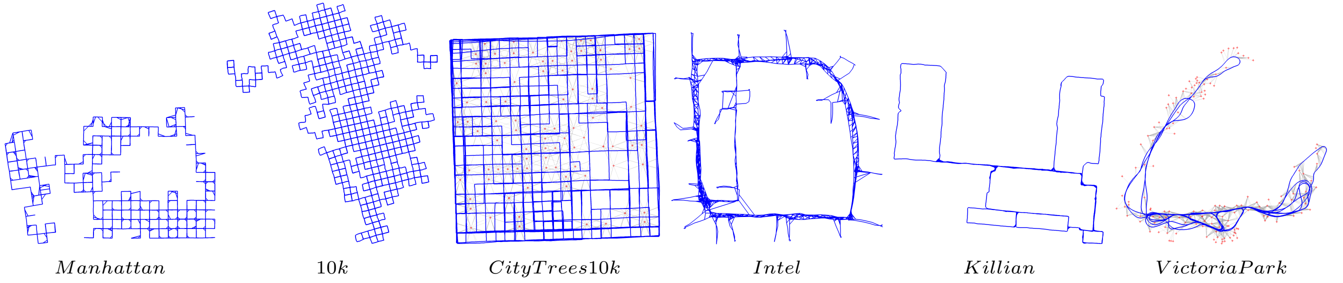

Fig. 3. The datasets used in our evaluations.

Table 1. Evaluation times of optimizers on multiple datasets (the best high accuracy solution times are in bold).

	Manhattan	10K	CityTrees10k	Intel	Killian	Victoria Park
SPA	23.8834	515.2880	n/a	1.4763	5.6260	n/a
$g2o$	94.9096	2134.3000	659.1590	5.0513	20.8899	293.1010
$iSAM$	64.5844	1768.8400	434.7500	4.4647	19.7519	209.1740
$iSAMb10$	9.9222	334.3650	60.2726	0.9442	3.6273	29.5268
$iSAMb100$	4.7142	289.7870	25.2429	1.3648	4.2522	12.6860
$allBatch - \Lambda$	10.0038	329.1840	**22.7070**	0.8424	**2.1485**	28.0194
$Inc - L$	**5.0274**	**183.3850**	25.5549	**0.7032**	2.5719	**16.0173**
$Inc - Lb10$	5.0275	166.7970	25.3064	0.6861	2.4637	14.6821

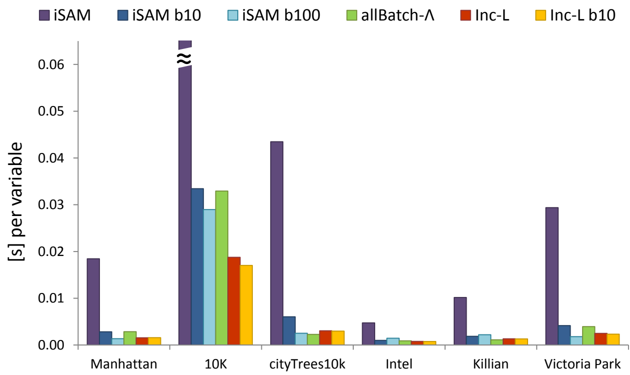

Fig. 4. Time comparison of multiple optimizers on datasets.

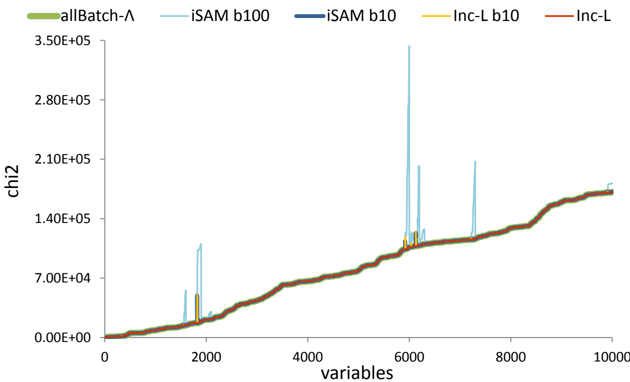

Fig. 5. Comparison of chi-squared errors.

iSAM2 Kaess et al. (2011, 2012), on the other hand, is an incremental algorithm based on gtsam library, and, at the time of writhing this paper the source code for iSAM2 was not available among the examples of the gtsam library. The reported results from iSAM2 papers Kaess et al. (2011, 2012) cannot be used for comparisons since they were measured on a radically different platform.

The evaluation was performed on three standard simulated datasets - *Manhattan*, Olson (2008), *10k* and *CityTrees10k*, Kaess et al. (2007) and three real datasets - *Intel*, Howard and Roy (2003), *Killian Court*, Bosse et al. (2004) and *Victoria park* dataset. The solution for each dataset is shown in Fig.3.

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, especially on smaller datasets.

Table 1 and Fig. 4 show the execution times of different implementations evaluated on the above mentioned datasets. The $b10$ and $b100$ flags represent the frequency of batch computations - after each 10 and after each 100 new added variables, respectively. For the results without those flags, the nonlinear system was solved every step in order to obtain the current estimation or, only when needed in the case of our new Incremental-L algorithm. Unlike g2o and SPA, iSAM and our implementation provide a solution every new update, even when the batch solver runs each 10 or each 100. This is an important characteristic for online applications. Therefore, and in order to make the visualization easy, Fig. 4 shows timing results only for the iSAM and our implementation.

All the times below the horizontal line in the table 1 are obtained using our implementation. The execution time of the Algorithm in Alg. 1 is indicated by $Inc - L$. The $Inc - Lb10$ is obtained by forcing batch every 10, but observe that this is not the natural way to execute our algorithm and has been introduced only for comparison purposes. $allBatch - \Lambda$ is a similar algorithm to the one introduced in Alg. 1 with the difference that it keeps and updates only the Λ matrix and performs matrix factorization every time a new linearization point needs to

be calculated. From the point of view of estimation quality, recalculating the system every time the linearization point changes, is the best the nonlinear solver can do but it can sometimes become computationally expensive. Even though, our optimized implementation performs very well also in the $allBatch - \Lambda$ case.

Figure 5 compares the quality of the estimations measured by the sum of squared errors - the χ^2 errors. The test was performed for the $10k$ dataset. Observe that our new algorithm, $Inc - L$ (in red in Fig. 5), nicely follows the $allBatch - \Lambda$ (in green in Fig. 5). Spikes appear when performing periodic batch solve in $iSAMb100$, $iSAMb10$ and $Inc - Lb10$ due to the fact that the error increases between the batch steps and drops afterwards.

As an overall remark, the $Inc - L$ has, in general, the best performance and provides very accurate results every step. Therefore, it is the most suitable implementation for online applications which involve nonlinear least squares solvers.

7. CONCLUSION

In this paper, we proposed a new, incremental least squares algorithm with applications to robotics. We targeted problems such as SLAM, which have a particular block structure, with the size of the blocks corresponding 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 by the comparison with the existing implementations on several standard datasets.

Even though the algorithm already proved efficient, several further improvements can be made. The current implementation does not allow for a fluid reordering of the variables, therefore the fill-in of the L factor is not the best we can obtain using well known heuristics such as AMD. This can be resolved by reordering the variables when performing Omega and Lambda-updates. Current implementation keeps the original variable order and performs reordering only when re-computing the entire L-factor.

The implementation itself could be improved by implementing Cholesky factorization by blocks in order to avoid the conversions between block and element-wise sparse matrix representations. Finally, 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 DSPs to clusters of GPUs.

REFERENCES

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

Davis, T.A. (2006). *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics.

Davis, T.A. and Hager, W.W. (1997). Modifying a sparse cholesky factorization.

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

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

K. Konolige, G. Grisetti, R.K.W.B.B.L. and R.Vincent (2010). Efficient sparse pose adjustment for 2d mapping. In *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 22–29.

Kaess, M., Ila, V., Roberts, R., and Dellaert, F. (2010). The Bayes tree: An algorithmic foundation for probabilistic robot mapping. In *Intl. Workshop on the Algorithmic Foundations of Robotics*.

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

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

Kaess, M., Ranganathan, A., and Dellaert, F. (2007). iSAM: Fast incremental smoothing and mapping with efficient data association. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 1670–1677. Rome, Italy.

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

Konolige, K. (2010). Sparse sparse bundle adjustment. In *British Machine Vision Conference*. Aberystwyth, Wales.

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

Olson, E. (2008). *Robust and Efficient Robot Mapping*. Ph.D. thesis, Massachusetts Institute of Technology.

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

Rosen, D., Kaess, M., and Leonard, J. (2012). An incremental trust-region method for robust online sparse least-squares estimation. In *IEEE Intl. Conf. on Robotics and Automation (ICRA)*, 1262–1269. St. Paul, MN.