AprilSAM: Real-time Smoothing and Mapping

Xipeng Wang Ryan Marcotte Gonzalo Ferrer Edwin Olson

***Abstract*— For online robots, incremental SLAM algorithms offer huge potential computational savings over batch algorithms. The dominant incremental algorithms are iSAM and iSAM2 which offer radically different approaches to computing incremental updates, balancing issues like 1) the need to relinearize, 2) changes in the desirable variable marginalization order, and 3) the underlying conceptual approach (i.e. the "matrix" story versus the "factor graph" story.)**

In this paper, we propose a new incremental algorithm that computes solutions with lower absolute error, and generally provides lower error solutions for a fixed computational budget than either iSAM or iSAM2. Key to AprilSAM's performance are a new dynamic variable reordering algorithm for fast incremental Cholesky factorizations, a method for reducing the work involved in backsubstitutions, and a new algorithm for deciding between incremental and batch updates.

I. Introduction

Simultaneous localization and mapping (SLAM) [1], [2], [3] is essential for a mobile robot to operate autonomously in a previously unknown environment. It provides continually updated estimates of both the position of the robot and the map of the environment. Online SLAM is an inherently incremental process: a robot repeatedly collects new information as it traverses its environment and incorporates that information into its map. An incremental SLAM solution should be both fast and accurate; speed ultimately determines the size of the environment that the robot can operate in within real-time performance constraints, while numerical accuracy is important for generating high-quality maps and position estimates. In this paper, we propose a real-time SLAM system, AprilSAM, that uses a new variable reordering algorithm coupled with fast incremental Cholesky factorization to improve system accuracy while maintaining real-time performance.

Smoothing methods formulate SLAM as a nonlinear least squares problem, which they solve to convergence by linearizing at a current estimate. Early smoothing methods (e.g. $\sqrt{\text{SAM}}$ [4]) employ a batch update at every step, making them too slow for real-time usage in large-scale maps. Incremental approaches such as iSAM [5] and iSAM2 [6] try to avoid batch updates, instead solving the least squares problem incrementally.

iSAM [5] uses matrix factorization for incremental updates and periodically performs batch updates. Because errors can accumulate without the linearization point being updated, iSAM's batch update does not guarantee that the nonlinear optimization converges to the optimal solution. Variable ordering has a large effect on performance as well: the computational cost of an incremental update can vary by an order of magnitude depending on the order in which variables are marginalized.

The authors are with the Computer Science and Engineering Department, University of Michigan, Ann Arbor, MI 48104, USA. {xipengw,ryanjmar,gferrerm,ebolson}@umich.edu

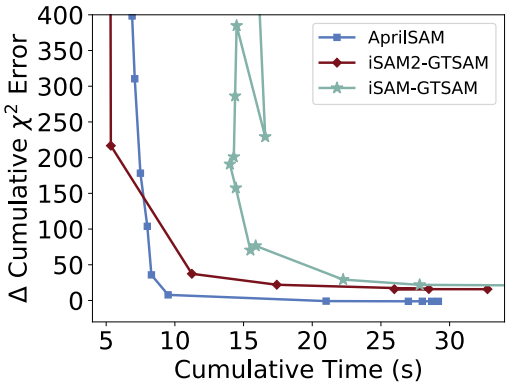


Fig. 1: Cost-Accuracy trade-off for iSAM-GTSAM, iSAM2-GTSAM and AprilSAM. All three algorithms have free parameters that trade off accuracy versus computational costs. The plot above shows the performance of these algorithms as the free parameter is varied. AprilSAM achieves the lowest absolute error and does so with generally lower computational times than both iSAM-GTSAM and iSAM2-GTSAM. However, iSAM2-GTSAM is faster for low-accuracy applications. Note that the relationship between the varied parameters and the cumulative time and χ^2 error is non-monotonic.

iSAM2 [6] introduces the Bayes tree to achieve incremental variable re-ordering and fluid re-linearization, eliminating the need for periodic batch updates. iSAM2 primarily focuses on improving running time, trading off some solution accuracy. Although iSAM2 elegantly connects matrix factorization with graphical model inference, implementing the Bayes tree is complex.

In this paper, we present an incremental SLAM approach based on fast incremental Cholesky factorization, which we call AprilSAM. Many of AprilSAM's core steps can be easily implemented using standard sparse linear systems libraries, which both eases implementation and allows highly optimized libraries to be used. In spite of this simplicity, it achieves state-of-the-art performance with highly accurate solutions.

The effectual contributions of AprilSAM are:

- Better absolute error than either iSAM or iSAM2 on benchmark datasets and
- Generally lower error for a given amount of computation time.

The technical contributions enabling these results include:

- A dynamic variable ordering algorithm that reduces fill-in and lowers the amount of computation expected in

subsequent incremental updates,
- An algorithm that provides criterion for selecting between incremental and batch updates
- A partial backsolve method that reduces the amount of computation involved in the incremental Cholesky decomposition.

II. Related Work

There has been much progress over the past decade to develop online SLAM solutions, much of which has been devoted to speeding up or avoiding altogether the batch updates of smoothing algorithms. We focus our review of related work on these types of approaches.

Kaess et al. [5] propose iSAM, an incremental smoothing and mapping algorithm that incrementally updates the QR factorization of the smoothing information matrix. To avoid unnecessary fill-in and accumulation of error, iSAM performs periodic variable re-ordering and re-linearization.

AprilSAM builds upon the foundational ideas of iSAM with some novel changes that lead to greater efficiency and higher accuracy. Although incremental updates are effective much of the time, a batch update is often necessary for the system to converge to the optimal solution. iSAM's periodic batch update does not account for accumulated state changes, leading to high-error solutions when a linearization point differs from the optimal solution. In this paper, we provide an algorithm that adaptively selects between incremental and batch updates (see Sec. IV-D). iSAM's incremental updates can become inefficient due to poor variable ordering, especially in the case of large loop closures. We propose a variable ordering algorithm that reduces the amount of computation needed for subsequent incremental updates (see Sec. IV-C). Finally, based on the evidence we provide in Sec. III, the QR factorization employed by iSAM induces more nonzero elements during the factorization process than the Cholesky factorization does, leading to slower computations. Motivated by this observation, we utilize an incremental Cholesky factorization rather than iSAM's QR factorization.

Kaess et al. [6] evolve the original iSAM to propose iSAM2, which achieves improvements in efficiency through incremental variable re-ordering and fluid re-linearization, eliminating the need for periodic batch steps. iSAM2 introduces the Bayes tree, a novel data structure for sparse nonlinear incremental optimization. Though iSAM2 elegantly connects matrix factorization with graphical model inference through the Bayes tree, its implementation is complex. In this paper, we present an incremental Cholesky factorization approach that can be easily implemented with standard sparse linear systems libraries, which eases implementation and allows highly optimized libraries to be used. iSAM2's fluid re-linearization and incremental re-ordering are used primarily to improve its running time, trading off some solution accuracy. As we show in Sec. V, AprilSAM yields solutions with lower absolute error as well as generally lower error given equivalent running time.

Polok et al. [7] present an incremental block Cholesky factorization for solving nonlinear least square problems. Its algorithm maintains both the smoothing information matrix and the factorization matrix while AprilSAM only keeps the latter one. Polok uses the CCOLAMD [8], which only considers the most recently accessed variables. However, we observe that computing variable orderings is hardly a bottleneck in this incremental process. Matrix reconstruction and the partial Cholesky decomposition take most of time, and their running time depends on variable ordering. In this paper, we propose a variable ordering algorithm for batch updates that anticipates possible upcoming loop closures. As we show in Sec. V, this allows for faster incremental updates in subsequent steps.

Kai et al. [9], [10] propose a novel batch algorithm for SLAM problems that distributes the workload in a hierarchical manner. They show how the original SLAM graph can be partitioned recursively into multiple-level submaps using the nested dissection algorithm [11], which leads to a cluster tree. By employing the nested dissection algorithm, their method greatly minimizes the dependencies between two subtrees, and the original SLAM graph can be optimized using a bottom-up inference along the corresponding cluster tree. In our paper, we do not explicitly apply the nested dissection algorithm to generate the cluster tree structure. Instead, we apply a min-heap based ordering method to allow faster incremental updates for large loop closures. Loop closures increase the connection degrees of nodes in a pose-graph, so they tend to be the separator point in the nested dissection algorithm. Our proposed algorithm also leverages this natural cluster tree structure, which only requires the update of a partial matrix in incremental factorization.

III. Problem Statement

In this section, we formulate Pose-SLAM as a least squares problem. Our goal is to estimate the robot states X given rigid transformation observations Z. According to Bayes law, the maximum likelihood solution is:

$$\begin{aligned} X_{ML} &= \arg\max_x P(X = x|Z = z) \\ &= \arg\max_x P(Z = z|X = x) \end{aligned} \tag{1}$$

If we assume a Gaussian measurement model, we can obtain X_{ML} by solving the the nonlinear least-squares problem

$$X_{ML} = \arg\min_x \sum_k \|\boldsymbol{z}_k - h(\boldsymbol{x}_{\boldsymbol{i}k}, \boldsymbol{x}_{\boldsymbol{j}k})\|^2_{\Omega_k} \tag{2}$$

where $\boldsymbol{z}_k$ is the k^{th} rigid transformation observation with corresponding information matrix Ω_k, $\boldsymbol{x}_{\boldsymbol{i}k}$ and $\boldsymbol{x}_{\boldsymbol{j}_k}$ are the nodes involved in that observation, and $h(\boldsymbol{x}_{\boldsymbol{i}k}, \boldsymbol{x}_{\boldsymbol{j}_k})$ is the expected rigid transformation based on the measurement model.

We linearize h as

$$h(\boldsymbol{x}_{\boldsymbol{i}k}, \boldsymbol{x}_{\boldsymbol{j}_k}) \approx h(\boldsymbol{\mu}_{\boldsymbol{i}k}, \boldsymbol{\mu}_{\boldsymbol{j}_k}) + \begin{bmatrix} J_k^{i_k}(\boldsymbol{\mu}_{\boldsymbol{i}k}) & J_k^{j_k}(\boldsymbol{\mu}_{\boldsymbol{j}_k}) \end{bmatrix} \begin{bmatrix} \delta\boldsymbol{x}_{\boldsymbol{i}k} \\ \delta\boldsymbol{x}_{\boldsymbol{j}_k} \end{bmatrix} \tag{3}$$

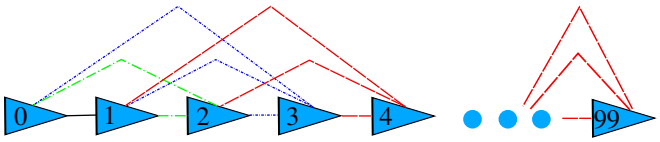


Fig. 2: A factor graph containing 100 (x, y, θ) nodes. A factor in the graph represents an observation of the rigid transformation between two nodes.

where $\boldsymbol{\mu}_{\boldsymbol{i}_k}$ is the current estimate for node $\boldsymbol{x}_{\boldsymbol{i}_k}$, and

$$J_k^{i_k}(\boldsymbol{\mu}_{\boldsymbol{i}_k}) = \frac{\delta h(\boldsymbol{x}_{\boldsymbol{i}_k}, \boldsymbol{x}_{\boldsymbol{j}_k})}{\delta \boldsymbol{x}_{\boldsymbol{i}_k}}|_{\boldsymbol{x}_{\boldsymbol{i}_k}=\boldsymbol{\mu}_{\boldsymbol{i}_k}} \tag{4}$$

with analogous definitions for $\boldsymbol{\mu}_{\boldsymbol{j}_k}$ and $J_k^{j_k}(\boldsymbol{\mu}_{\boldsymbol{j}_k})$. Applying this linearization to (2), we obtain the least squares formulation

$$\delta_X = \arg\min_{\delta_x} \|A\delta_x - b\|^2 \tag{5}$$

where $A = \Omega^{\frac{1}{2}} J$ and $b = \Omega^{\frac{1}{2}}(z - h(\mu_i, \mu_j))$.

Many methods exist to solve (5), with the most common direct approach being matrix factorization. In iSAM [5], Kaess solves the linear system using QR factorization as

$$QR\delta_x = b \tag{6}$$

where $A = QR$. In contrast, we utilize the following Cholesky factorization to solve (5) by solving two triangular matrix equation.

$$R^T R\delta_x = b^* \tag{7}$$

where $A^* = A^T A = R^T R$ and $b^* = A^T b$.

Both factorization methods (Cholesky [12] and QR [13]) compute the same matrix R. In the GraphSLAM problem, however, QR factorization induces many more non-zero elements during this computation than Cholesky factorization does, slowing down the sparse matrix math involved in the computation. GraphSLAM problems have more factors than nodes, making A (used by QR) a tall matrix and $A^T A$ (used by Cholesky) a relatively small square matrix.

We tested both factorization methods on the graph shown in Fig. 2. This graph has 100 (x, y, θ) nodes, each of which is connected to the two nodes ahead of itself. In Fig. 3, we show the number of nonzero elements at each process step for QR factorization using the Householder method and for Cholesky factorization. Even though the number of nonzero elements is equal at the end of process, the QR factorization induces more nonzero elements at each step, leading to longer computation time.

Motivated by this observation, we utilize an incremental Cholesky factorization approach instead of the incremental QR factorization of Kaess [5]. Though QR is expected to produce more accurate results due to its lower condition number, we find the difference to be negligible in practice.

IV. Approach

In this section, we present AprilSAM, the incremental SLAM algorithm shown in Alg. 1. AprilSAM updates R and y incrementally, a process we describe in Sec. IV-A and Sec. IV-B. Each time AprilSAM performs a batch update, we apply a dynamic re-ordering algorithm as described in Sec. IV-C. The algorithm AprilSAM uses to select between batch and incremental updates is listed in Sec. IV-D.

Moving forward, we make a small notational update for the sake of clarity, dropping the * from A and b. That is, A^* and b^* become A and b, respectively.

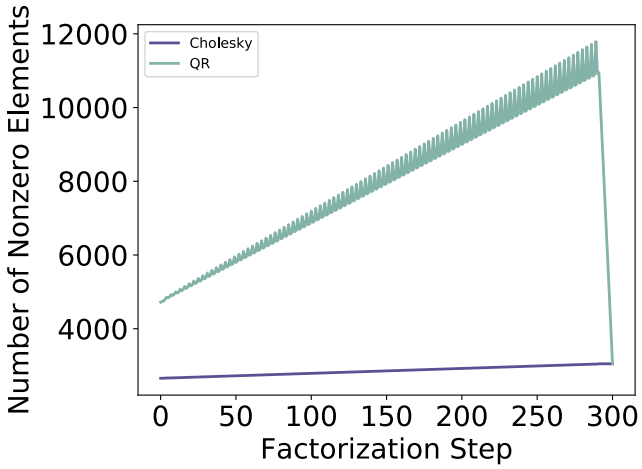


Fig. 3: The number of nonzero elements at each step of the matrix factorization for the graph shown in Fig. 2. QR factorization induces more nonzero elements than Cholesky during the factorization process, leading to slower computations.

Algorithm 1 AprilSAM

function AprilSAM(Graph G, NewFactors f, R, y, NL_T, δ_T, Δ_T)
  $G \leftarrow$ AugmentGraph(G, f)
  $(R, y) \leftarrow$ IncrementalUpdate(G)
  $\delta_x \leftarrow$BackwardSubstitute(R, y)
  $L \leftarrow \emptyset$ ▷ Set of nodes to be linearized
  for all nodes $x_i \in G$ **do**
    $x_i \leftarrow x_i + \delta_{x_i}$
    if $\delta_{x_i} \geq \delta_T$ **then**
      $L \leftarrow L \cup x_i$
  $n_{lc} \leftarrow$CountPossibleLoopClosureNodes(G)
  if $n_{lc}\times$ RunTime(incremental) $\geq$ RunTime(batch) or $|L| \geq NL_T$ or $\|\delta_x\| \geq \Delta_T$ **then**
    for all nodes $x_i \in G$ **do**
      UpdateLinearizationPoint(x_i)
    $(R, y) \leftarrow$ BatchUpdate(G)
    $\delta_x \leftarrow$BackwardSubstitute(R, y)
    for all nodes $x_i \in G$ **do**
      $x_i \leftarrow x_i + \delta_{x_i}$
end function

A. Incremental Cholesky Decomposition

Whereas Polok [7] maintains both the information matrix A and the factorization matrix R, our algorithm only requires

the latter. Recall the forms of these two matrices:

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \quad R = \begin{bmatrix} R_{00} & R_{01} \\ \mathbf{0} & R_{11} \end{bmatrix} \tag{8}$$

An intermediate matrix R^{in} of the decomposition process has the form

$$R^{in} = \begin{bmatrix} R_{00} & R_{01} \\ \mathbf{0} & A_{11}^{in} \end{bmatrix} \tag{9}$$

where A_{11}^{in} is the filled-in version of submatrix A_{11} resulting from the partial factorization. Given the completed factorization R, we can compute A_{11}^{in} as

$$A_{11}^{in} = R_{11}^T R_{11} \tag{10}$$

This in turn allows us to reconstruct the intermediate matrix R^{in}.

Let $\tilde{A}$ be the evolved version of A following the addition of new information. This update only affects a portion A_{11} of the information matrix, with corresponding effects to R_{11} in the factorization. That is, the new information matrix $\tilde{A}$ and its factorization matrix $\tilde{R}$ have the forms:

$$\tilde{A} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} + A_{new} \end{bmatrix} \quad \tilde{R} = \begin{bmatrix} R_{00} & R_{01} \\ \mathbf{0} & \tilde{R}_{11} \end{bmatrix} \tag{11}$$

Because $\tilde{A} = \tilde{R}^T \tilde{R}$, we can express the updated portion of the information matrix as

$$\tilde{R}_{11}^T \tilde{R}_{11} = A_{11}^{in} + A_{new} \tag{12}$$

Applying (10), we can re-write (12) in terms of only the previous factorization matrix R and the new information A_{new}. That is,

$$\tilde{R}_{11}^T \tilde{R}_{11} = R_{11}^T R_{11} + A_{new} \tag{13}$$

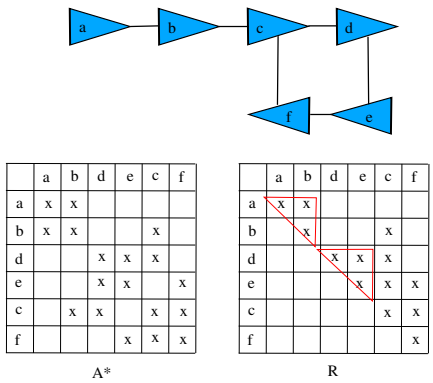
In Fig. 4a, we show an example of a batch update. By ordering nodes c and f at the end of matrix, we separate the graph into small clusters. This cluster structure accelerates the incremental update process.

Fig. 4b shows an example of adding an odometry edge. Since the new node g is only connected to node f, only columns f and g of R will be changed.

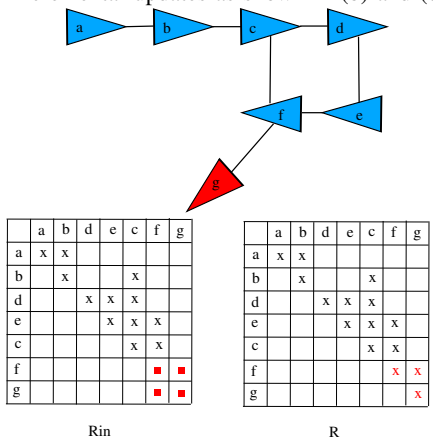
Fig. 4c shows an example of adding a loop closure. The new node g is connected to nodes f and b. Due to the cluster structure, nodes d and e are separated with loop closure node b, and those columns of R are not changed.

B. Solving the Triangular Matrix Equations

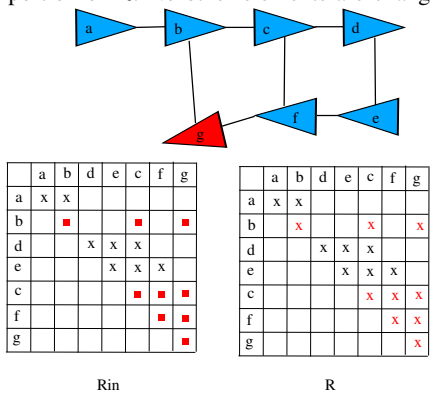
The Cholesky factorization of (7) can be used to solve for δ_x by successive forward and backward substitution. Namely, we solve $R^T y = b$ by forward substitution and use the result to solve $R\delta_x = y$ by backward substitution. As we argued in the previous section, only a portion of R and b change when performing an incremental update. We now show that we do not need to maintain b and solve the first equation every step. Instead, we can update y incrementally and only solve the second equation, $R\delta_x = y$.



(a) An example of batch update with min-heap based ordering. By ordering node c and node f at the end of matrix, the graph is separated into small clusters. This cluster structure allows for faster incremental updates as shown in (b) and (c).



(b) An example of adding an odometry edge. Since the new node g is only connected to node f, only column f and g of R will be changed. The red squares represent the reconstructed portion of intermediate matrix A^{in} plus the new information A_{new}, as detailed in (12). The red crosses represent the modified portion of R. No other elements are changed.



(c) An example of adding loop closure. The new node g is connected to nodes f and b. Due to the cluster structure, nodes d and e are separated by loop closure node b, so those columns of R are not changed.

Fig. 4: Incremental update of R

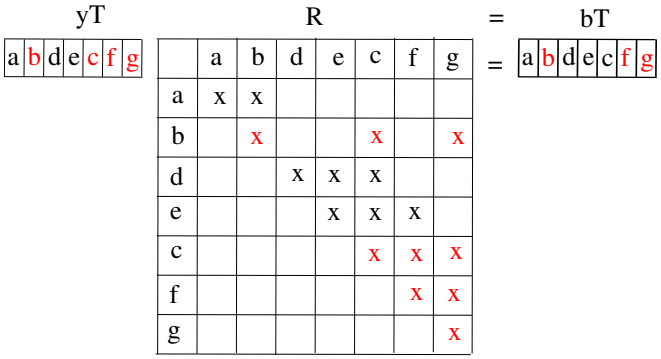


Fig. 5: Incremental update of $\boldsymbol{y}$. Since only nodes b and f are connected to the newly added node g, in the residual vector $\boldsymbol{b}$, only b, f, and g will change. When we solve this linear system, we only need to solve b, c, f, and g in the $\boldsymbol{y}$ due to the parallel triangular structure. We mark the affected elements in red.

Recall the forms of R^T, y, and b:

$$R^T y = b$$
$$\begin{bmatrix} R_{00}^T & \mathbf{0} \\ R_{01}^T & R_{11}^T \end{bmatrix} \begin{bmatrix} y_{00} \\ y_{10} \end{bmatrix} = \begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix} \tag{14}$$

When information is added, only a portion of the RHS changes. That is,

$$\tilde{b} = \begin{bmatrix} \tilde{b}_{00} \\ \tilde{b}_{10} \end{bmatrix} = \begin{bmatrix} b_{00} \\ b_{10} + b_{new} \end{bmatrix} \tag{15}$$

From (14), we obtain the equation for the affected portion of b:

$$b_{10} = R_{01}^T y_{00} + R_{11}^T y_{10} \tag{16}$$

Applying (15), we write the corresponding equation for the modified portion of $\tilde{b}$ as

$$\begin{aligned} \tilde{b}_{10} &= \tilde{R}_{01}^T \tilde{y}_{00} + \tilde{R}_{11}^T \tilde{y}_{10} \\ b_{10} + b_{new} &= R_{01}^T y_{00} + \tilde{R}_{11}^T \tilde{y}_{10} \end{aligned} \tag{17}$$

Finally, substituting the value of b_{10} from (16), we obtain an update step that allows us to solve y incrementally:

$$\tilde{R}_{11}^T \tilde{y}_{10} = R_{11}^T y_{10} + b_{new} \tag{18}$$

In Fig. 5, we mark the affected portions of matrix and vectors as red. Similar to the update of R in the incremental Cholesky decomposition, we only need to maintain $\boldsymbol{y}$ and reconstruct and solve for the affected portion each time. In this case, we do not need to re-calculate parts of $\boldsymbol{y}$ that are not changed.

C. Min-heap based ordering

In this paper, we present a min-heap based node ordering algorithm based on BHAMD [14]. Compared to the original BHAMD algorithm, this ordering has less non-zero fill-ins because it continually checks the scores of a removed node's neighbors and adds them back to correct list. This prevents the case in which a node with large initial score remains in the list even when its score is reduced because of node removal.

Algorithm 2 Min-heap Ordering Procedure

```
Create a set of lists where each list is associated with a score and contains nodes having same scores.
Add all lists into a min heap: MH.
while MH is not empty do bestList = getMinList from MH minScore = MH.getMinKey
    for each node nd in bestList do
        if nd.score ≤ minScore then
            remove nd
            Update nd's score.
            for each node nnd in nd's neighbors do
                if nnd.score ≤ minScore then
                    push it back into the bestList
                else
                    push it back into the correct list
        else
            push it back into the correct list
```

In addition, in this min-heap based algorithm, the score of a node not only depends on its connection degree but also its distance to the most recently added node. We can predict which nodes may be included in upcoming loop closures based on connections between nodes and the radius at which loop closures are considered. Our ordering algorithm will order these nodes at the end of matrix automatically based on their score.

Alg. 2 details the ordering algorithm. We begin by adding lists to the min-heap, where each list is associated with a possible node score. We then add each node to the proper list based on its score.

At each iteration, we extract from the heap the list that is associated with the minimum node score. We consider each node from this list in turn, along with their neighbors. Nodes whose score no larger than the current minimum score are eliminated. Nodes with larger scores are moved to the correct list based on their scorinto the list associated with that score.

The score of a node is given by

$$score = \begin{cases} N + \frac{1}{r}, & r < R_T \\ n, & else \end{cases} \tag{19}$$

where r is the distance to the most recently added node, R_T is the threshold inside of which loop closures are considered, n is the connection degree of the node, and N is the number of nodes in the graph.

D. Incremental update vs. Batch update

Re-linearization in a batch update can reduce system error, but batch updates are typically more computationally expensive than incremental updates. iSAM attempts to handle this dilemma by performing batch updates at regular intervals. In contrast, AprilSAM performs batch updates when one of three conditions is met (see also in Alg. 1):

1) AprilSAM tracks nodes that have changed significantly in a set $L = \{x_i : \delta_{x_i} > \delta_T\}$. If enough nodes have undergone significant changes (i.e. $|L| > NL_T$), AprilSAM performs a batch update.

2) AprilSAM performs a batch update if the norm of the total state changes becomes too large (i.e. $\|\delta_x\| > \Delta_T$). Since the SLAM nonlinear least square problem is solved by repeatedly solving linear equations, this condition keeps the current solution from diverging too far from the optimal solution.
3) Due to the cost associated with reconstructing $R_{11}^T R_{11}$ in (13), incremental updates may actually take more time than batch updates when a large loop closure happens. This is especially true for poor variable orderings. In such a case, a batch update could save time by ordering variables involved in possible loop closures at the end, making subsequent incremental updates much faster. When the estimated time to execute an incremental update (i.e. the running time of the previous incremental update multiplied by the number of nodes involved in possible loop closures) is larger than the running time of the most recent batch update, AprilSAM chooses to perform a batch update.

V. Evaluation

In this section, we evaluate AprilSAM on real-world benchmark datasets. We compare AprilSAM's performance to that of iSAM [5] and iSAM2 [6], which represents the current state-of-the-art. We also evaluate several variants of AprilSAM to gain greater insight into its performance.

A. Methodology

$\sqrt{\text{SAM}}$-Chol performs a batch update every step using the Cholesky decomposition, providing us with a performance baseline against which to compare the incremental approaches. We introduce here several variants of AprilSAM used in the evaluation to isolate each component of the system:

- *AprilSAM-FullSolve* removes AprilSAM's mechanism for reducing the computation involved in solving the triangular matrix equations (Sec. IV-B), instead solving the full set of equations each time.
- *AprilSAM-NoPredictiveOrder* replaces the variable reordering algorithm (Sec. IV-C) with one that simply orders the most recent graph node last and applies BHAMD [14] to the remaining nodes.
- *AprilSAM-Periodic* removes the algorithm for selecting between incremental and batch updates (Sec. IV-D), instead performing batch updates periodically.

We use the authors' implementation of iSAM2 in GTSAM [15], labeled as iSAM2-GTSAM. We tested the algorithms on two benchmark datasets: W10000 [16] and M3500 [17]. W10000 contains 10000 (x, y, θ) nodes and 64311 (x, y, θ) factors. M3500 contains 3500 nodes and 5453 factors.

B. Evaluation on W10000 and M3500

Figs. 6 and 7 show the performance of the tested algorithms on the W10000 dataset and M3500 dataset, respectively.

Figs. 6a and 7a show the cumulative running time of the tested algorithms. AprilSAM runs faster than AprilSAM-NoPredictiveOrder, which supports our claim that the min-heap based variable ordering algorithm saves running time.

Figs. 6b and 7b show the χ^2 error of the algorithms relative to $\sqrt{\text{SAM}}$-Chol, which performs a batch update at each step. Recall the distinction between AprilSAM and AprilSAM-Periodic: AprilSAM uses the criteria listed in Alg. 1 to decide between incremental and batch updates, whereas iSAM-Periodic performs batch updates periodically. We configured iSAM-Periodic to perform a batch update every 100 steps for the W10000 dataset and every 50 steps for the M3500 dataset. AprilSAM has a faster running time and higher accuracy than iSAM-Periodic, suggesting that these decision criteria are effective.

The negative error in Figs. 6b and 7b is a result of AprilSAM detecting large state changes after an incremental update and then performing a batch update immediately. This situation is similar to performing a batch update twice when calculating χ^2 error based on state, though the linearization point is only changed once right before the batch update.

Figs. 6c and 7c show the cumulative density function of the distribution of iteration running times. On the W10000 dataset, AprilSAM runs in less than 10 ms on 70 percent of its iterations and less than 100ms on 93 percent. On the M3500 dataset, AprilSAM achieves these same metrics in 93 percent and 100 percent of iterations, respectively.

The incremental algorithms can trade off running time for accuracy, a property which we evaluate in Figs. 6d and 7d. In the case of iSAM2-GTSAM, we exchange running time for accuracy by varying the value of the parameter β, which determines how many graph nodes are re-linearized [6]. Specifically, we use $\beta = \{1, 0.9, \ldots, 10^{-1}, 10^{-2} \ldots, 10^{-6}\}$. Though AprilSAM has more free parameters than iSAM2-GTSAM, we only vary the parameter δ_T (see Alg. 1), setting its value equal to iSAM2-GTSAM's β. When we examine the cumulative running time and relative cumulative χ^2 square error in Figs. 6d and 7d, we observe that beyond a certain point, AprilSAM always maintains a lower error given the same amount of running time. In case of iSAM-GTSAM, we use batch update interval as $\{100, 90, \cdots, 10, 5, 1\}$. AprilSAM is better than iSAM-GTSAM in terms of both running time and error. We do not show all points of iSAM-GTSAM and iSAM2-GTSAM because their errors are too large to fit on the plot. In contrast, with the same parameters, AprilSAM's errors are relatively small. This is because its decision to use batch or incremental updates depends not just on state changes, but on the tracked difference in running time between both types of updates.

For small enough parameter settings (which lead to higher running times), both iSAM2-GTSAM and AprilSAM converge to particular values of χ^2 error. AprilSAM significantly outperforms iSAM2-GTSAM in terms of accuracy at this convergence point.

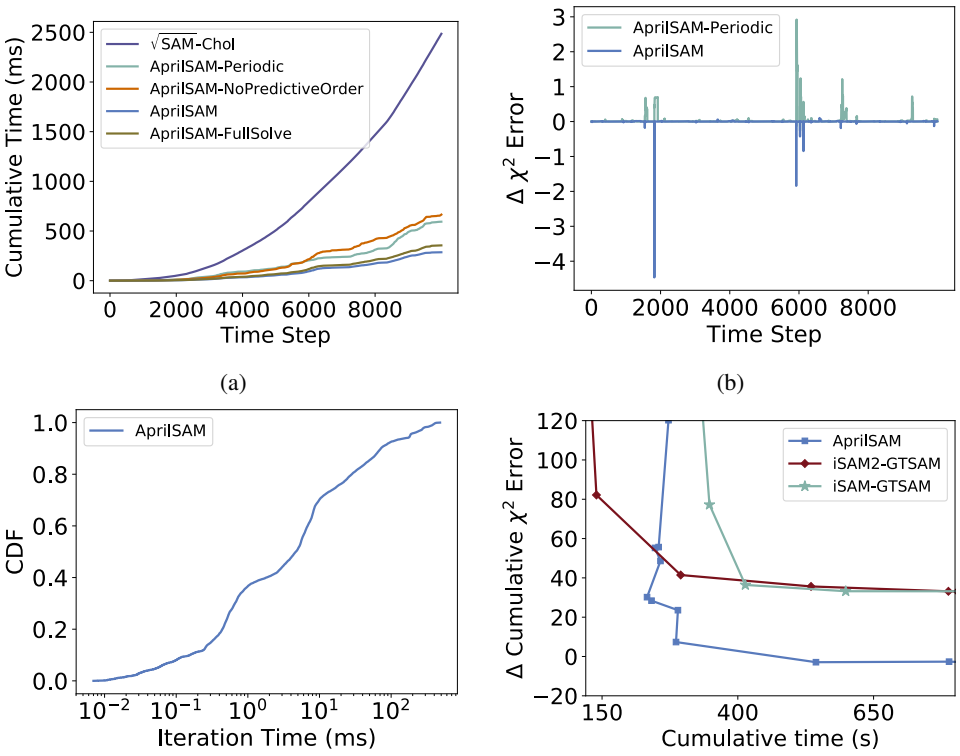


Fig. 6: Evaluation on W10000 dataset

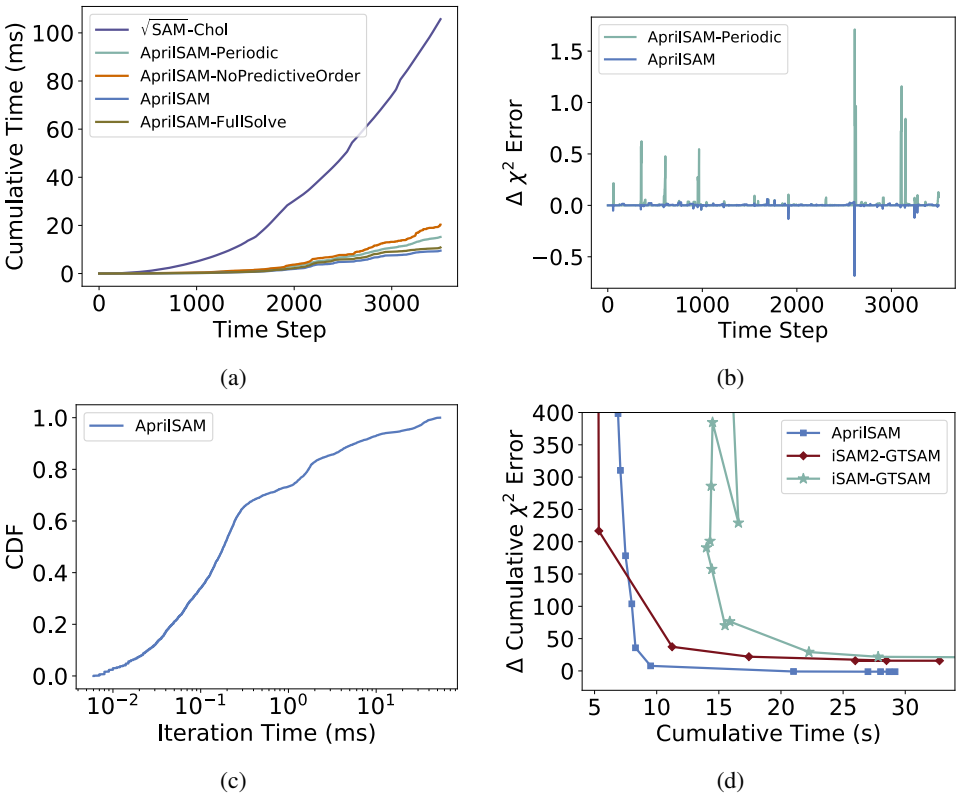


Fig. 7: Evaluation on M3500 dataset

VI. Conclusion

In this work, we introduce an incremental SLAM algorithm, AprilSAM, that uses a min-heap based variable reordering algorithm coupled with fast incremental Cholesky factorization. This algorithm drives down system error while maintaining real-time performance. We evaluate AprilSAM on real-world data, comparing it with the current state-of-the-art algorithm, iSAM2. We have shown that in spite of its simplicity, AprilSAM achieves better absolute error than other algorithms while generally having higher accuracy for a given amount of computation.

References

[1] C. Cadena, L. Carlone, H. Carrillo, Y. Latif, D. Scaramuzza, J. Neira, I. Reid, and J. J. Leonard, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, 2016.

[2] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I," *IEEE robotics & automation magazine*, vol. 13, no. 2, pp. 99–110, 2006.

[3] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (SLAM): Part II," *IEEE Robotics & Automation Magazine*, vol. 13, no. 3, pp. 108–117, 2006.

[4] F. Dellaert and M. Kaess, "Square Root SAM: Simultaneous localization and mapping via square root information smoothing," *The International Journal of Robotics Research*, vol. 25, no. 12, pp. 1181–1203, 2006.

[5] M. Kaess, A. Ranganathan, and F. Dellaert, "iSAM : Incremental Smoothing and Mapping," *IEEE Transactions on Robotics*, 2008.

[6] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, "iSAM2 : Incremental Smoothing and Mapping with Fluid Relinearization and Incremental Variable Reordering," *The International Journal of Robotics Research*, 2012.

[7] L. Polok, V. Ila, M. Solony, P. Smrz, and P. Zemcik, "Incremental Block Cholesky Factorization for Nonlinear Least Squares in Robotics," in *Robotics: Science and Systems (RSS)*, 2013.

[8] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, "A column approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 3, pp. 353–376, 2004.

[9] K. Ni, D. Steedly, and F. Dellaert, "Tectonic SAM: Exact, out-of-core, submap-based SLAM," in *Robotics and Automation, 2007 IEEE International Conference on*, IEEE, 2007.

[10] K. Ni and F. Dellaert, "Multi-level submap based SLAM using nested dissection," 2010.

[11] R. E. Tarjan and M. Yannakakis, "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs," *SIAM Journal on computing*, vol. 13, no. 3, pp. 566–579, 1984.

[12] Golub, Gene H and Van Loan, Charles F, *Matrix computations*, vol. 3. JHU Press, 2012.

[13] W. Gander, "Algorithms for the QR decomposition," *Res. Rep*, vol. 80, no. 02, pp. 1251–1268, 1980.

[14] P. Agarwal and E. Olson, "Variable reordering strategies for SLAM," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2012.

[15] F. Dellaert, "Factor graphs and GTSAM: A hands-on introduction," tech. rep., Georgia Institute of Technology, 2012.

[16] 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*, pp. 27–30, 2007.

[17] E. Olson, J. Leonard, and S. Teller, "Fast iterative alignment of pose graphs with poor initial estimates," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2262–2269, 2006.