Efficient Modification of the Upper Triangular Square Root Matrix on Variable Reordering

Khen Elimelech[1], and Vadim Indelman[2]

***Abstract*—In probabilistic state inference, we seek to estimate the state of an (autonomous) agent from noisy observations. It can be shown that, under certain assumptions, finding the estimate is equivalent to solving a linear least squares problem. Solving such a problem is done by calculating the upper triangular matrix $\boldsymbol{R}$ from the coefficient matrix $\boldsymbol{A}$, using the QR or Cholesky factorizations; this matrix is commonly referred to as the "square root matrix". In sequential estimation problems, we are often interested in periodic optimization of the state variable order, e.g., to reduce fill-in, or to apply a predictive variable ordering tactic; however, changing the variable order implies expensive re-factorization of the system. Thus, we address the problem of modifying an existing square root matrix $\boldsymbol{R}$, to convey reordering of the variables. To this end, we identify several conclusions regarding the effect of column permutation on the factorization, to allow efficient modification of $\boldsymbol{R}$, without accessing $\boldsymbol{A}$ at all, or with minimal re-factorization. The proposed parallelizable algorithm achieves a significant improvement in performance over the state-of-the-art incremental smoothing and mapping approach, which considers incremental factorization on updates.**

***Index Terms*—Probabilistic Inference; SLAM; Incremental Least Squares; Parallel Algorithms; Sparse Systems.**

I. Introduction

A. Background

To operate reliably, autonomous agents and robots must model their state in the world; however, most often these agents can only rely on noisy measurements of their environment. This problem, known as probabilistic state inference, is a fundamental concern in the fields of Artificial Intelligence and Robotics [1]. An important instance of this problem is Simultaneous Localization And Mapping (SLAM), in which we wish to estimate an agent's location, while mapping its surroundings. Such online systems are of particular interest, as the solution must be computed in real time. In recent years, smoothing-based solution approaches have been growing in popularity, due to their robustness over traditional filter-based solutions. These approaches, and most prominently factor-graph-based Smoothing and Mapping (SAM) [2], utilize graphical models for constraint satisfaction. Now, to understand the context of our problem of interest, we must first briefly formalize this state inference problem.

Manuscript received: October 16, 2020; Accepted December 13, 2020.

This paper was recommended for publication by Editor Javier Civera upon evaluation of the Associate Editor and Reviewers' comments. This work was partially supported by the Israel Science Foundation.

[1]Khen Elimelech is with the Robotics and Autonomous Systems Program, Technion – Israel Institute of Technology khen@technion.ac.il

[2]Vadim Indelman is with Department of Aerospace Engineering, Technion – Israel Institute of Technology vadim.indelman@technion.ac.il

Digital Object Identifier (DOI): see top of this page.

Let the vector $\boldsymbol{X} \doteq [x_1, \ldots, x_n]^T$ represent the current state of our agent. Given a set of probabilistic constraints $f_i(\boldsymbol{X})$ (also known as factors), representing the noisy measurements, we can define the belief over $\boldsymbol{X}$ as $\mathbb{P}(\boldsymbol{X}) = \prod_i f_i(\boldsymbol{X})$. We would like to find the best estimate of $\boldsymbol{X}$ given the measurements, i.e., the maximum-a-posteriori (MAP) estimate. If the factors are linear, and the belief is Gaussian, it can be shown (see [2]) that finding the MAP estimate of $\boldsymbol{X}$ corresponds to solving a least square problem. In the least squares approach to linear regression, we seek an assignment for a vector $\boldsymbol{X} \in \mathbb{R}^n$, which minimizes the squared error on a set linear constraints. A system of m linear constraints can be written in a matrix form, where every row represents a linear equation:

$$\boldsymbol{A} \cdot \boldsymbol{X} = \boldsymbol{b}, \quad \boldsymbol{A} \in \mathbb{R}^{m \times n}. \tag{1}$$

Note that if the factors are not linear, we can still use this solution approach, by utilizing an iterative local linearization. We assume the system is over-determined, i.e., containing no less variables than equations ($m \geq n$), and $\boldsymbol{A}$ to be of full column rank n. Under these assumptions, the solution is unique. To find this solution in a stable manner, we can use the upper triangular factor $\boldsymbol{R}$, which is provided by finding the QR factorization (decomposition) of $\boldsymbol{A}$, such that $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$, for an orthogonal matrix $\boldsymbol{Q}$. Equivalently, $\boldsymbol{R}$ can be derived from the Cholesky decomposition of the symmetric and positive semi-definite information matrix $\boldsymbol{\Lambda} \doteq \boldsymbol{A}^T\boldsymbol{A}$, such that $\boldsymbol{\Lambda} = \boldsymbol{R}^T\boldsymbol{R}$; this matrix is hence known as the "square root matrix" of $\boldsymbol{\Lambda}$. If we require the diagonal entries of $\boldsymbol{R}$ to be non-negative, then the two factorizations provide the same matrix $\boldsymbol{R}$ [3]. Using $\boldsymbol{R}$, we can conveniently find the estimate of $\boldsymbol{X}$ via back/front substitution; for more details, see [4].

B. Motivation

In sequential estimation problems, as time progresses, new measurements are collected; these may lead to addition of new constraints, or extension of the state with new variables. At each such event, we shall appropriately update $\boldsymbol{R}$, in order to refine our estimate. To efficiently keep up with these updates, we are often interested in optimizing the order of state variables. For example, we may want to utilize a fill-reducing order. When calculating the QR factorization of $\boldsymbol{A}$ (in order to derive $\boldsymbol{R}$), we access it column-by-column in a process known as variable elimination [4]. The order of columns in $\boldsymbol{A}$ (which matches the order of state variables) affects the fill-in and sparsity pattern of $\boldsymbol{R}$, and, by such, may affect the computational cost of subsequent updates, and the memory footprint. Although finding the optimal fill-reducing order is

NP-complete [5], various heuristics, such as COLAMD [6], provide empirically good results. Problem-specific heuristics, e.g., for SLAM [7], which exploit known state topology, are also applicable. We may even consider alternative ("non-fill-reducing") variable orders. For example, autonomous agents are often asked to plan their future actions, based on their current belief (commonly referred to as "belief space planning"). According to the Predictive Incremental Variable Ordering Tactic (PIVOT) [8], by predicting the future state evolution according to candidate actions, we can optimize the variable order in the current state, and improve the solution efficiency.

Unfortunately, while new constraints can often be incrementally and efficiently incorporated, changing the order of variables modifies the factor $\boldsymbol{R}$ in a non-trivial way, and implies re-factorization of the entire system. The computational cost of such operation is, hence, exceptionally high.

C. Problem Definition and Contribution

We formulate the problem of interest as follows: Consider the matrix $\boldsymbol{A}$, and its upper triangular factor $\boldsymbol{R}$, given by the QR factorization, which satisfies $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{R}$ for an orthogonal matrix $\boldsymbol{Q}$. Also consider a variable permutation p, with a matching column permutation matrix $\boldsymbol{P}$. We wish to find the upper triangular factor $\boldsymbol{R_p}$ of the permuted matrix $\boldsymbol{A_p} \doteq \boldsymbol{A}\boldsymbol{P}$, which satisfies $\boldsymbol{A_p} = \boldsymbol{Q_p}\boldsymbol{R_p}$ for an orthogonal matrix $\boldsymbol{Q_p}$. As explained, we may equivalently consider $\boldsymbol{R}$ and $\boldsymbol{R_p}$ to be the Cholesky factors of $\boldsymbol{\Lambda}$ and $\boldsymbol{\Lambda_p} \doteq \boldsymbol{P}^T\boldsymbol{\Lambda}\boldsymbol{P}$, respectively.

To clarify, we do not wish to choose a new variable order; we consider it to be given, and address how to perform the appropriate modification to $\boldsymbol{R}$ practically and efficiently. To achieve this goal, we derive three key contributions: first, we identify that variable reordering only requires a local modification of $\boldsymbol{R}$, in a contained block of rows; second, we explain how to identify independent row blocks in $\boldsymbol{R}$, which can be modified independently, and in parallel; third, we explain how to efficiently derive the orthogonal transformation, which defines the required modification for each row block. We then utilize these three ideas in two algorithmic variants: one, by manipulating the factor directly, without accessing $\boldsymbol{A}$; and second, with partial re-factorization of $\boldsymbol{A}$. Each of the variants holds specific advantages, as we shall compare.

Besides the aforementioned contribution to real-time operation of autonomous agents, the introduced algorithms are also especially valuable for "large-scale" optimization problems (e.g., offline map optimization over extremely large areas), in which full re-factorization is often practically impossible.

D. Related Work

Numerous methods (e.g., [9], [10], [11]) examined the modification of $\boldsymbol{R}$ on low-rank updates and downdates, i.e., the addition or removal of rows from/to $\boldsymbol{A}$. However, column permutation is not classified as a low-rank change, making such modification methods inapplicable.

A recent work [12] tried to address a similar problem. There, the authors provided symbolic formulas for swapping the order of two consecutive variables. They showed that in such instance, only the corresponding pairs of rows and columns in the square root matrix are affected, while the rest of the matrix entries remain unchanged. They then proposed to use the formulas in a "bubble sort"-like algorithm, to apply permutations involving more variables. However, this method is only suitable for simple permutations, as for extensive permutations, this "bubble sort" might require up to $O(n^2)$ swaps. The authors even declared that beyond a certain amount of such swaps, one should simply resort back to $\boldsymbol{\Lambda}$, and re-apply the Cholesky factorization on the permuted matrix. Another drawback of that approach is that every such swap requires to explicitly update all the affected elements. Thus, for extensive permutations, some rows might be completely recalculated over and over. As to be explained, our algorithm is built to avoid this very problem.

At the arrival of new constraints, the incremental smoothing and mapping algorithm (iSAM [13]) performs incremental modification to $\boldsymbol{R}$, instead of re-computing it "from scratch", and without accessing the matrix $\boldsymbol{A}$. However, such updates do not involve variable reordering, and the algorithm does not manage to avoid periodic re-factorization of the *entire* system, in order to optimize the variable order. Other incremental methods, such as iSAM2 [14] and SLAM++ [15], incorporate the new constraints by performing partial re-factorization of $\boldsymbol{A}$ or $\boldsymbol{\Lambda}$ – starting the elimination from the first variable to receive a new constraint, according to the variable order. During this re-factorization, these methods are able to reorder the re-eliminated variables; e.g., using a fill-reducing order on these variables, for efficient calculation of the update. Such reordering helps to maintain the sparsity of $\boldsymbol{R}$, as desired. However, as we are only able to reorder a subset of the variables during updates, the total variable order may not be globally optimal. Hence, it is still sensible to periodically conduct a "standalone" reordering process, while considering all the state variables. Yet, we are not aware of any approaches which allow modification of $\boldsymbol{R}$ on global variable reordering, without completely re-factorizing the system.

As mentioned, we also utilize smart partial re-factorization in one of the two variants of the proposed modification algorithm; but, as to be seen, in comparison to the aforementioned incremental methods, our algorithm requires re-factorization of a significantly smaller sub-matrix of $\boldsymbol{A}$.

We note that, like our approach, modern "multi-frontal" implementations of the QR factorization (e.g., [16], [17]) also utilize parallelization, by applying multiple transformations (e.g., rotations) at once. However, generally speaking, the parallel fronts are not independent; these fronts dynamically change, and new fronts are initiated according to the results of former transformations. Such multi-frontal implementation requires intricate synchronization. Our algorithm, on the other hand, yields a high-level block division of the rows, such that, by definition, there are no dependencies between the blocks, and there is no need for communication nor synchronization among them; each block of rows can theoretically be processed on a different machine. This approach is also bandwidth efficient, as we do not initiate factorization on the entire matrix, but only process blocks of it at a time. Under this division, each block can be factorized using any implementation of QR, including the aforementioned multi-frontal ones.

II. PRELIMINARIES

Formally, the permutation p is a bijective function $\{1,\dots,n\} \rightarrow \{1,\dots,n\}$, which defines a mapping of each state variable from index i to index $p(i)$. Such permutation can be described with an orthogonal permutation matrix:

$$\boldsymbol{P} \doteq \begin{bmatrix} | & & | \\ \boldsymbol{e_{p(1)}} & \cdots & \boldsymbol{e_{p(n)}} \\ | & & | \end{bmatrix}, \tag{2}$$

where $\boldsymbol{e_i} \in \mathbb{R}^n$ is the i-th unit vector. Hence, the permuted coefficient matrix (with permuted columns) is given as $\boldsymbol{AP}$.

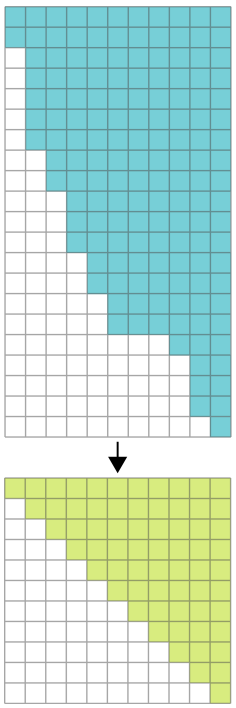

Figure 1: $\boldsymbol{A}$ (top), and $\boldsymbol{R}$, given from $\mathtt{qr}\{\boldsymbol{A}\}$.

For visualization purposes, we consider an exemplary least squares system, which will be used as a running example throughout the paper. In the top sub-figure of Fig. 1, we can see the original coefficient matrix $\boldsymbol{A}$ of the system, and underneath it, its factor given from $\mathtt{qr}\{\boldsymbol{A}\}$. We wish to apply the variable permutation p, such that

$$[\,1,\dots,11\,]\cdot\boldsymbol{P} \doteq [\,1,2,4,3,8,9,5,7,6,10,11\,]. \tag{3}$$

Each of the sub-figures represents the sparsity pattern of the corresponding matrix; colored cells represent entries which are (possibly) Non-Zero (NZ); different colors correspond to different classification of the entries, as to be described. For convenience, the rows of $\boldsymbol{A}$ are sorted according to the column-index of their leading NZ entry (i.e, first NZ in the row); nonetheless, $\boldsymbol{R}$ is invariant under row reordering in $\boldsymbol{A}$. We do not make any assumptions on the density (or sparsity) of these matrices.

Finally, let us explain the notation used in this paper: the operation $\mathtt{qr}\{\times\}$ returns the QR factorization; $\mathtt{blkdiag}\{\times,\dots,\times\}$ returns a matrix with the given inputs on its block diagonal; $\boldsymbol{I_n}$ is the identity matrix of size n; $i_1:i_2$ marks the vector $[i_1,...,i_2]$; end marks the last index; $\boldsymbol{M}(\mathcal{I},\mathcal{J})$ is the sub-matrix of elements $(i,j): i\in\mathcal{I}, j\in\mathcal{J}$ of $\boldsymbol{M}$; $\boldsymbol{M_1M_2}(\times,\times)$ marks the block of $\boldsymbol{M_1M_2}$, and $\boldsymbol{M_1}\cdot\boldsymbol{M_2}(\times,\times)$ marks the block of $\boldsymbol{M_2}$.

III. DIRECT MODIFICATION

A. The Local Effect of Variable Reordering

First, we shall try to derive $\boldsymbol{R_p}$ by appropriately modifying the original square root matrix $\boldsymbol{R}$, and without accessing $\boldsymbol{A}$. Let us look at $\boldsymbol{RP}$, i.e., $\boldsymbol{R}$ with columns permuted according to p. As visualized in the left sub-figure in Fig. 2, applying the column permutation breaks the triangular shape of the original root matrix $\boldsymbol{R}$. We may re-apply $\mathtt{qr}$ to "correct" it:

$$\{\boldsymbol{Q'},\boldsymbol{R'}\} \doteq \mathtt{qr}\{\boldsymbol{RP}\} \tag{4}$$

The permutation does not affect the rank of the matrix, and since $\boldsymbol{R}$ is of full rank (the rows/columns are linearly independent), so is $\boldsymbol{RP}$. Therefore, we know a *unique* $\mathtt{qr}$ factorization of it exists. A similar re-application of $\mathtt{qr}$ to "correct" the factor is performed also by the aforementioned iSAM algorithm; though, there, the factor is modified due to the addition of constraints (new rows), and not permutation of the state variables.

This factorization can be calculated in several manners. We may choose to apply a series of Givens rotations $\{\boldsymbol{Q_k}\}_k$ to zero all elements below the diagonal, one entry at a time, column-by-column. Each of these rotations is applied only on a pair of rows in the factorized matrix; the angle of rotation is chosen such that the leading NZ entry in the lower of the two rows is zeroed (for more details, see [3]). Since the factorization exists, we know that after a finite number of such rotations, we will end up with an upper triangular matrix $\boldsymbol{R'}$ with a non-zero diagonal, such that

$$\boldsymbol{R'} = \boldsymbol{Q'}^T\cdot(\boldsymbol{RP}), \text{ where } \boldsymbol{Q'}^T \doteq \prod_k \boldsymbol{Q_k}. \tag{5}$$

Surely, since the product of orthogonal matrices remains orthogonal, $\boldsymbol{R'}$ is the desired "square root matrix" of the permuted system:

$$\boldsymbol{A_p} = (\boldsymbol{QR})\cdot\boldsymbol{P} = \boldsymbol{Q}\cdot(\boldsymbol{RP}) = \boldsymbol{Q}\cdot(\boldsymbol{Q'R'}) = (\boldsymbol{QQ'})\cdot\boldsymbol{R'}. \tag{6}$$

Although it provides the solution, we do not suggest to use this naive approach to modify $\boldsymbol{R}$, as it may be far from optimal. Yet, due to the uniqueness of $\boldsymbol{R'}$, we know that this $\boldsymbol{R'} \equiv \boldsymbol{R_p}$, and we can use this solution to derive general properties $\boldsymbol{R_p}$, which we will exploit to derive it more efficiently.

Let us mark with j_{first} and j_{last} the the minimal and maximal non-fixed points of p, respectively; i.e., the minimal and maximal indices $j \in 1:n$, for which $p(j) \neq j$. NZ entries below the diagonal can only appear in $\boldsymbol{RP}$ in rows indexed between j_{first} and j_{last}. Hence, application of the aforementioned Givens rotations shall only modify these rows of the factorized matrix; $\boldsymbol{R'}$ ($\equiv \boldsymbol{R_p}$) remains equal to $\boldsymbol{RP}$ in all other rows. The underlying conclusion, which holds regardless of the manner by which $\boldsymbol{R_p}$ is calculated, is conveyed in Proposition 1.

Proposition 1 (The Local Effect of Variable Reordering)**.** *The block of rows $j_{first} : j_{last}$ of $\boldsymbol{R_p}$ relies only on the respective block of rows of $\boldsymbol{R}$; the matrices may differ in value only in these rows. The top $(j_{first}-1)$ rows, and bottom $(n-j_{last})$ rows of $\boldsymbol{R_p}$, are identical to the respective rows of $\boldsymbol{R}$ (up to column permutation).*

Thus, $\boldsymbol{R_p}$ may be derived using any $\mathtt{qr}$ implementation as:

$$\boldsymbol{R_p} = \begin{bmatrix} — & \boldsymbol{R'}_{\text{top}} & — \\ \boldsymbol{0} & \boldsymbol{R'}_{\text{middle}} & \\ — & \boldsymbol{R'}_{\text{bottom}} & — \end{bmatrix}, \text{ where} \tag{7}$$

$$\boldsymbol{R'}_{\text{top}} \doteq \boldsymbol{RP}(1:(j_{\text{first}}-1), 1:n), \tag{8}$$

$$\{\boldsymbol{Q'}_{\text{middle}}, \boldsymbol{R'}_{\text{middle}}\} \doteq \mathtt{qr}\{\boldsymbol{RP}(j_{\text{first}}:j_{\text{last}}, j_{\text{first}}:n)\}, \tag{9}$$

$$\boldsymbol{R'}_{\text{bottom}} \doteq \boldsymbol{RP}((j_{\text{last}}+1):end, 1:n). \tag{10}$$

Accordingly, $\boldsymbol{Q'} \doteq \mathtt{blkdiag}\{\boldsymbol{I}_{j_{\text{first}}-1}, \boldsymbol{Q'}_{\text{middle}}, \boldsymbol{I}_{m-j_{\text{last}}}\}$ marks the "correcting" orthogonal matrix, such that $\boldsymbol{Q_p} = \boldsymbol{Q}\cdot\boldsymbol{Q'}$ completes the solution. This "localized" direct modification method is visualized in Fig. 2.

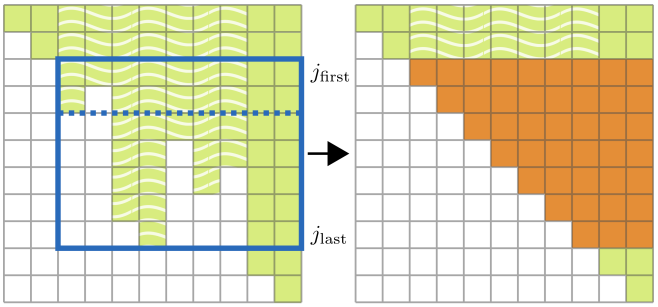

Figure 2: The local effect of variable reordering. Blue borders mark the re-factorized sub-matrix; green-colored entries represent entries of $\boldsymbol{R}$; orange represents entries calculated from re-factorization of $\boldsymbol{RP}$; white overlay indicates column permutation. Modification of $\boldsymbol{R}$ only requires factorization of the block of rows $j_{\text{first}} : j_{\text{last}}$ of $\boldsymbol{RP}$. Using Algorithm 1, we can divide this row block (dashed blue line) into the two sub-blocks, which can be factorized independently.

We may also reach a similar conclusion from probabilistic analysis. The belief $\mathbb{P}(\boldsymbol{X})$ (which we defined in Section I-A) can be factorized to a product of conditional probabilities:

$$\mathbb{P}(\boldsymbol{X}) = \prod_{k=1}^{n-1} \mathbb{P}(x_k | \pi(x_k)) \cdot \mathbb{P}(x_n), \tag{11}$$

where $\pi(x_j)$ denotes the set the variables x_j is conditioned upon, for a given variable elimination order; a variable may only be conditioned on those which follow it according to that order. Now, say we wish to modify this factorization to match a permuted elimination order, which is defined by the permutation p. By definition, $\pi(x_j) = \pi(x_{p(j)})$ for $j < j_{\text{first}}$ and $j > j_{\text{last}}$. Thus,

$$\mathbb{P}(\boldsymbol{X}) = \prod_{j=j_{\text{first}}}^{j_{\text{last}}} \mathbb{P}(x_{p(i)} | \pi(x_{p(i)})) \cdot \ldots \prod_{\substack{1 \le j < j_{\text{first}} \\ j_{\text{last}} < j \le (n-1)}} \mathbb{P}(x_k | \pi(x_k)) \cdot \mathbb{P}(x_n) \tag{12}$$

Meaning, the permutation only affects the conditional probabilities of the variables $j_{\text{first}} : j_{\text{last}}$, or, in other words, permutation has only a local effect on the belief factorization – with no assumptions on the type of distribution! Nonetheless, we recall that, when $\boldsymbol{X}$ is normally distributed, this factorization of $\mathbb{P}(\boldsymbol{X})$ corresponds to factorization of $\boldsymbol{A}$ to the matrix $\boldsymbol{R}$, where each conditional probability corresponds to the respective row of $\boldsymbol{R}$ [2]. Hence, in accordance with Proposition 1, we may again conclude that applying the permutation only leads to modification of rows $j_{\text{first}} : j_{\text{last}}$ of this matrix.

B. Identifying Independent Row Blocks

It is known that every permutation can be decomposed into disjoint cycles. Each cycle defines a "sub-permutation" which only affects a distinct subset of the input elements (in our case, the indices $1 : n$), while all other elements are stationary (i.e., mapped to themselves). We can write our exemplary permutation (Eq. 3) in "cycle notation":

$$p = (1)(2)(3\,4)(5\,8\,7)(6\,9)(10)(11). \tag{13}$$

Each set of parentheses represents a disjoint cycle; each number in the cycle is mapped to the one that follows it, and the last number is mapped back to the first one.

Thanks to Proposition 1, we know that applying each such "sub-permutation" only requires factorization of a specific block of rows of $\boldsymbol{RP}$, encapsulated between the minimal and maximal index in the cycle, independently of the other rows. However, we note that multiple "sub-permutations" cannot always be applied independently of each other, as these blocks of rows, corresponding to each of the cycles, are not always distinct. This happens when the cycles define overlapping ranges, like $(5\,8\,7)$ and $(6\,9)$. In this case, the two cycles cannot be considered independently, and we have to factorize the entire block of rows $5 : 9$. While accounting for such scenarios, Algorithm 1 specifies how to divide the rows of $\boldsymbol{RP}$ into distinct row blocks, which can be treated independently, given a certain permutation. The result of this algorithm is visualized in Fig. 2, where, given p, we identified two row blocks of $\boldsymbol{RP}$. After the blocks are identified, we can "correct" each of them by applying an independent `qr` factorization. These factorizations can (and should) be executed *in parallel*.

Algorithm 1: Identifying Independent Row Blocks

Inputs:
Permutation p

Output:
$block\_division$ – a set of vectors, each containing row (variable) indices of an independent row block in $\boldsymbol{R_p}$

```
block_division ← {}
j1 ← 1
while j1 ≤ n do
    j2 ← p(j1)
    if j2 > j1 then
        // not a stationary point, look for
           intersecting cycles
        k ← j1 + 1
        while k ≤ j2 do
            j2 ← max{j2, p(k)}
            k ← k + 1
    block_division ← block_division ∪ {j1 : j2}
    j1 ← j2 + 1
```

C. Efficient Row Modification

Assume we wish to factorize the non-triangular block of rows $j_1 : j_2$ of $\boldsymbol{RP}$, in order to derive the respective block of rows of $\boldsymbol{R_p}$. We note that, even without the leading zero columns, this is a "wide" matrix, i.e., the number of columns is greater than or equal the number of rows. Hence, let us examine a sub-matrix of this row block, containing only its first $l \doteq j_2 - j_1 + 1$ NZ columns: $square\_block \doteq \boldsymbol{RP}(j_1 : j_2, j_1 : j_2)$. This is a square block taken around the diagonal of $\boldsymbol{RP}$. This block is of full rank l, as it differs from the block $\boldsymbol{R}(j_1 : j_2, j_1 : j_2)$ only in the order of columns, and the latter is upper triangular with NZs on its diagonal (and hence of full rank). Therefore, $square\_block$ has a unique factorization:

$$\{\boldsymbol{Q}'_{\text{block}}, square\_block'\} = \texttt{qr}\{square\_block\}, \tag{14}$$

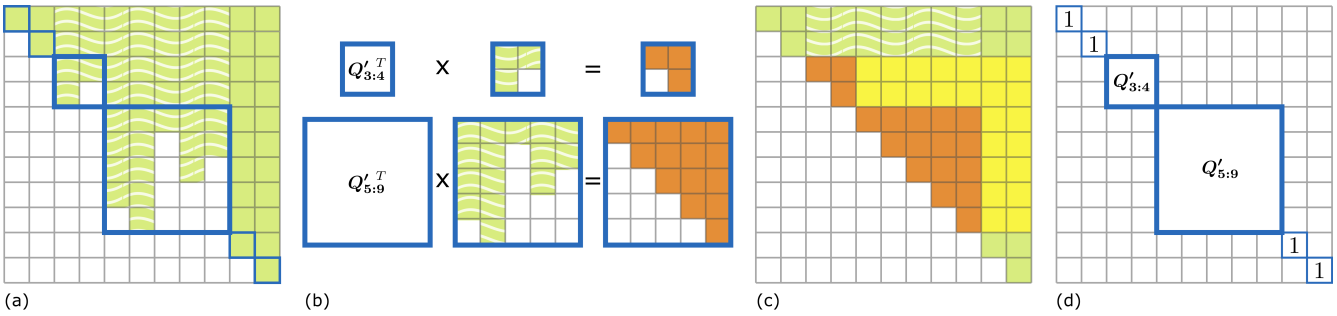

Figure 3: Applying Algorithm 2 for optimized direct modification of the factor $\boldsymbol{R}$ given p. Blue borders mark the re-factorized sub-matrices; green-colored entries represent entries of $\boldsymbol{R}$; orange represents entries calculated from re-factorization of $\boldsymbol{RP}$; yellow represent entries calculated by applying a pre-calculated transformation; white overlay indicates column permutation. (a) $\boldsymbol{RP}$ and its block diagonal structure. (b) Applying `qr` independently on each square block. (c) Deriving $\boldsymbol{R_p}$ by applying the calculated orthogonal transformations on the remaining elements in the respective rows. (d) The "correcting" orthogonal matrix $\boldsymbol{Q'}$ follows the same block structure.

such that $square\_block'$ is also an upper triangular matrix with NZs on its diagonal. Thus, it is sufficient to examine $square\_block$, in order to derive the matrix $\boldsymbol{Q'}_{\text{block}}$, which defines the desired transformation for the entire row block.

This approach essentially separates the calculation and application of this orthogonal transformation; this can improve the modification efficiency in comparison to a "naive" application of `qr` on the entire row block. Calculating the `qr` factorization requires finding and applying an appropriate series of transformations (e.g., Givens rotations or Householder reflections [3]), which together form the orthogonal matrix. Generally, we must apply each rotation, in order to find the next one in the series. This means that each entry in the factorized matrix is modified multiple times throughout the factorization, according to the number of transformations needed. Thus, by only factorizing the square block, and not the entire rows, we can avoid unneeded calculations. The entries beyond column j_2 (i.e., $\boldsymbol{RP}(j_2+1:n)$) do not need to be continuously modified throughout the factorization process; they can be modified only once, by multiplying them in the matrix $\boldsymbol{Q'}_{\text{block}}{}^{T}$ after it is found.

Overall, we shall repeat this process for every row block of $\boldsymbol{RP}$ which requires factorization (provided by Algorithm 1); by doing so, we can identify a block diagonal structure, such that all the NZ entries of $\boldsymbol{RP}$ below the diagonal are contained (as visualized in Fig 3a). In order to calculate $\boldsymbol{R_p}$, we shall (independently) factorize each of these square blocks, and apply the transformations to the rest of the respective row blocks. As mentioned, $\boldsymbol{Q_p} = \boldsymbol{Q} \cdot \boldsymbol{Q'}$, and $\boldsymbol{Q'}$ can be found by concatenating each block's $\boldsymbol{Q'}_{\text{block}}$ matrix, according to the block diagonal structure. Algorithm 2 summarizes the approach.

Algorithm 2: Optimized Factor Modification

Inputs:
- Upper triangular factor $\boldsymbol{R}$
- Permutation p with a matching column permutation matrix $\boldsymbol{P}$
- **For re-factorization:** $\boldsymbol{A}$ and $marginals$ (see Section IV-A)

Output:
- Modified factor $\boldsymbol{R_p}$

$\boldsymbol{RP} \leftarrow \boldsymbol{R} \cdot \boldsymbol{P}$
$\boldsymbol{AP} \leftarrow \boldsymbol{A} \cdot \boldsymbol{P}$; `// *only for re-factorization*`
$\boldsymbol{R_p} \leftarrow zeros(n,n)$; `// init. matrix`
$block\_division \leftarrow$ run Algorithm 1 on p
foreach $block\_indices$ **in** $block\_division$ **do** `// in parallel`
  $j_1 \leftarrow block\_indices(1)$
  $l \leftarrow length\{block\_indices\}$
  if $l == 1$ **then**
    $\boldsymbol{R_p}(j_1, j_1:n) \leftarrow \boldsymbol{RP}(j_1, j_1:n)$
  else
    $j_2 \leftarrow block\_indices(end)$
    if *DIRECT MODIFICATION* **then**
      $row\_block \leftarrow \boldsymbol{RP}(j_1:j_2, j_1:n)$
    else if *RE-FACTORIZATION* **then**
      $row\_block \leftarrow \begin{bmatrix} marginals_p\{j_1-1\} \\ \boldsymbol{AP}(\mathtt{I}_{\boldsymbol{AP}}(j_1:j_2), j_1:n) \end{bmatrix}$
    `// calculate transformation`
    $square\_block \leftarrow row\_block(1:end, 1:l)$
    $\{Q'_{\text{block}}, square\_block'\} \leftarrow \mathtt{qr}\{square\_block\}$
    `// apply transformation`
    $rows' \leftarrow Q'_{\text{block}}{}^{T} \cdot row\_block(1:end, (l+1):end)$
    `// assign in the modified matrix`
    $\boldsymbol{R_p}(j_1:j_2, j_1:j_2) \leftarrow square\_block'(1:l, 1:end)$
    $\boldsymbol{R_p}(j_1:j_2, (j_2+1):end) \leftarrow rows'(1:l, 1:end)$

D. Numerical Concerns

We recall that $\boldsymbol{R_p}$ is unique (under the aforementioned conditions), and, thus, all modification methods should theoretically yield the same matrix. However, the orthogonal transformations used in the factorization are operations with real numbers, which are inevitably rounded, in order to be computed in a machine with limited precision. Therefore, two mathematically equivalent sequences of operations might, in practice, reach different results, due to slight numerical errors.

Altogether, we can divide the modification methods into two categories: direct modification of $\boldsymbol{R}$, without re-accessing $\boldsymbol{A}$ (as considered thus far); and methods which, to varying capacity, involve re-factorization of $\boldsymbol{A}$, under the new variable order. Direct modification tends to be faster, as it takes advantage of the original factorization, in which we handled the "heavy load" of reducing the overdetermined system to a "square" one. When performing direct modification, we essentially apply additional transformations *on top* of those which were

applied in the original factorization: $\boldsymbol{R_p} = \boldsymbol{Q'}^T \cdot \boldsymbol{Q}^T \cdot \boldsymbol{AP}$. Hence, as a rule of thumb, this approach corresponds to an overall longer (although theoretically equivalent) series of operations to reach $\boldsymbol{R_p}$, than if we were to re-factorize $\boldsymbol{AP}$ ($\boldsymbol{A} \rightarrow \boldsymbol{R} \rightarrow \boldsymbol{R_p}$ vs. $\boldsymbol{A} \rightarrow \boldsymbol{R_p}$). This may cause the matrix returned by direct modification to be more prone to numerical errors, and, most importantly, to "false" fill-in. Such errors occur when a series of operations that should accumulate to a zero entry, yields a slightly different value. This added fill-in can be problematic in sparse systems or sequential tasks, as it can accumulate over time. As each of the approaches has its advantages (as we shall compare), we follow the previous discussion by addressing modification via re-factorization.

IV. Modification via Re-factorization

Clearly, as demonstrated in Fig. 4a, we can derive $\boldsymbol{R_p}$ by simply re-factorizing the system under the new variable order:

$$\{\boldsymbol{Q_p}, \boldsymbol{R_p}\} = \mathtt{qr}\{\boldsymbol{AP}\}. \tag{15}$$

In the top sub-figure, we can see the permuted coefficient matrix $\boldsymbol{A_p}$, and in the bottom one – its triangular factor. Yet, this "naive" solution is far from efficient; as we can see, this solution is oblivious to the original factorization, and all entries of $\boldsymbol{R_p}$ are calculated from scratch. As we shall formulate, it is possible to take advantage of the original factorization, and perform smart partial re-facatorization of $\boldsymbol{A_p}$.

A. Marginal Factor Caching

When calculating $\mathtt{qr}\{\boldsymbol{A}\}$, we build $\boldsymbol{R}$ in ascending order of rows, by gradually applying orthogonal transformations on the rows of $\boldsymbol{A}$ [3]. We start by factorizing all the rows in which the leading NZ is in the first column. By the end of this step, this block of rows from $\boldsymbol{A}$ is reduced to a single row with the leading NZ in the first column (which is the first row of $\boldsymbol{R}$), and "marginal factors"; these are rows which were not present in $\boldsymbol{A}$ before, and will not appear in $\boldsymbol{R}$ – they represent intermediate steps in the factorization, and shall be processed in the next factorization steps. We then move on to the next block of rows, with the leading NZs in the second column (considering also the marginal factors generated in the previous steps), and so on, until all the rows of $\boldsymbol{R}$ are calculated. We know that the j-th row of $\boldsymbol{R}$ is calculated once there is no more than a single row with leading NZ in the j'-th column, for each $j' \leq j$. Now, assume that after finding $\boldsymbol{R}$, we are interested in the factor $\tilde{\boldsymbol{R}}$ of another matrix $\tilde{\boldsymbol{A}} \in \mathbb{R}^{\tilde{m} \times n}$.

Let $\mathtt{I}_{\boldsymbol{A}}\{j_1 : j_2\}$ mark the set of indices of rows of $\boldsymbol{A}$, in which the leading NZ is between the j_1-th and j_2-th columns, and further assume the block of rows $\mathtt{I}_{\boldsymbol{A}}(1 : j_{\text{first}} - 1)$ of $\boldsymbol{A}$ is equal to the block of rows $\mathtt{I}_{\tilde{\boldsymbol{A}}}(1 : j_{\text{first}} - 1)$ of $\tilde{\boldsymbol{A}}$. From the previous discussion, we understand that the calculation of the top $j_{\text{first}} - 1$ rows of $\boldsymbol{R}$ and $\tilde{\boldsymbol{R}}$ is equivalent. Thus, we can potentially take advantage of the prior factorization efforts (of $\boldsymbol{A}$), to efficiently factorize $\tilde{\boldsymbol{A}}$: given access the marginal factors, we can "restart" the prior factorization from an intermediate step, after the top $j_{\text{first}} - 1$ rows of $\boldsymbol{R}$ were calculated, and continue factorizing rows $\mathtt{I}_{\tilde{\boldsymbol{A}}}(j_{\text{first}} : n)$ of $\tilde{\boldsymbol{A}}$, to derive the remaining rows of $\tilde{\boldsymbol{R}}$. Surely, we can take advantage of this incremental calculation even if the overlapping rows between $\boldsymbol{A}$ and $\tilde{\boldsymbol{A}}$ differ in the order of columns $j_{\text{first}} : j_{\text{last}}$; when restarting the factorization, the top $j_{\text{first}} - 1$ rows of $\boldsymbol{R}$ (and the marginal factors) can just be permuted accordingly, and shall remain "triangular". Of course, this idea is only applicable if we save (or cache) the marginal factors while calculating the prior factorization. This requires a custom implementation of the `qr` algorithm, e.g., as described in Algorithm 3.

This idea, is not, in fact, a novel contribution of this paper; it is prominently used in state-of-the-art approaches for incremental smoothing and mapping, such as the aforementioned iSAM2, which performs incremental updates to $\boldsymbol{R}$, at the arrival of new constraints (i.e., addition of rows to $\boldsymbol{A}$). Although we do not consider addition of constraints, but only reordering of $\boldsymbol{A}$'s columns, we can still utilize this idea here, to efficiently calculate $\boldsymbol{R_p}$ from $\boldsymbol{AP}$, as visualized in Fig. 4b. Note that typically, when using cached marginals, formation of the orthogonal matrices ($\boldsymbol{Q_p}$ and $\boldsymbol{Q}$, in Algorithms 2 and 3, respectively) from their blocks requires tedious index tracking. However, once $\boldsymbol{R}$ or $\boldsymbol{R_p}$ are found, the orthogonal matrices can be easily found, if needed, by calculating $\boldsymbol{Q} = \boldsymbol{A} \cdot \boldsymbol{R}^{-1}$, or $\boldsymbol{Q'} = \boldsymbol{RP} \cdot \boldsymbol{R_p}^{-1}$ (where $\boldsymbol{Q_p} = \boldsymbol{Q} \cdot \boldsymbol{Q'}$), respectively.

Algorithm 3: `qr` with Marginal Factor Caching

Inputs:
- $\boldsymbol{A}$

Output:
- Upper triangular factor $\boldsymbol{R}$
- $marginals$ – a set of matrices, describing the marginal factors created after eliminating each variable

$\boldsymbol{R} \leftarrow zeros(n, n)$; // init. matrix
$marginals \leftarrow \{\}$; // init. empty set
$Marginal \leftarrow zeros(0, n)$; // init. empty matrix
for j **to** 1 **to** n **do**
  $\{\boldsymbol{Q}_{\text{block}}, \boldsymbol{R}_{\text{block}}\} \leftarrow \mathtt{qr}\left\{\begin{bmatrix} Marginal \\ \boldsymbol{A}(\mathtt{I}_{\boldsymbol{A}}(j), j : n) \end{bmatrix}\right\}$
  $\boldsymbol{R}(j, j : n) \leftarrow \boldsymbol{R}_{\text{block}}(1, 1 : end)$; // j-th row of $\boldsymbol{R}$
  $Marginal \leftarrow \boldsymbol{R}_{\text{block}}(2 : end, 2 : end)$
  $marginals\{j\} \leftarrow Marginal$

B. Optimized Re-factorization

Nonetheless, this method is not optimized for pure variable permutation, as we consider here; we can improve it by utilizing our previous conclusions. From Proposition 1, we know that not only the top rows of $\boldsymbol{R}$ are preserved in the permutation, but also the bottom ones. So, similarly to Eq. 9, we can also "skip" the factorization of rows $\mathtt{I}_{\boldsymbol{AP}}(j_{\text{last}}+1 : n)$ of $\boldsymbol{AP}$, and factorize only rows $\mathtt{I}_{\boldsymbol{AP}}(j_{\text{first}} : j_{\text{last}})$, which, together with the permuted marginal factors, yield rows $j_{\text{first}} : j_{\text{last}}$ of $\boldsymbol{R_p}$, i.e.,

$$\mathtt{qr}\left\{\begin{bmatrix} marginals_p\{j_{\text{first}} - 1\} \\ \boldsymbol{AP}(\mathtt{I}_{\boldsymbol{AP}}(j_{\text{first}} : j_{\text{last}}), j_{\text{first}} : n) \end{bmatrix}\right\}. \tag{16}$$

This improvement is visualized in Fig. 4c. We note that here, unlike in direct modification, it is possible that the triangular block resulting from this re-factorization would contain additional rows, representing residual marginals factors; such rows can be trimmed, to be left only with the relevant rows of $\boldsymbol{R_p}$.

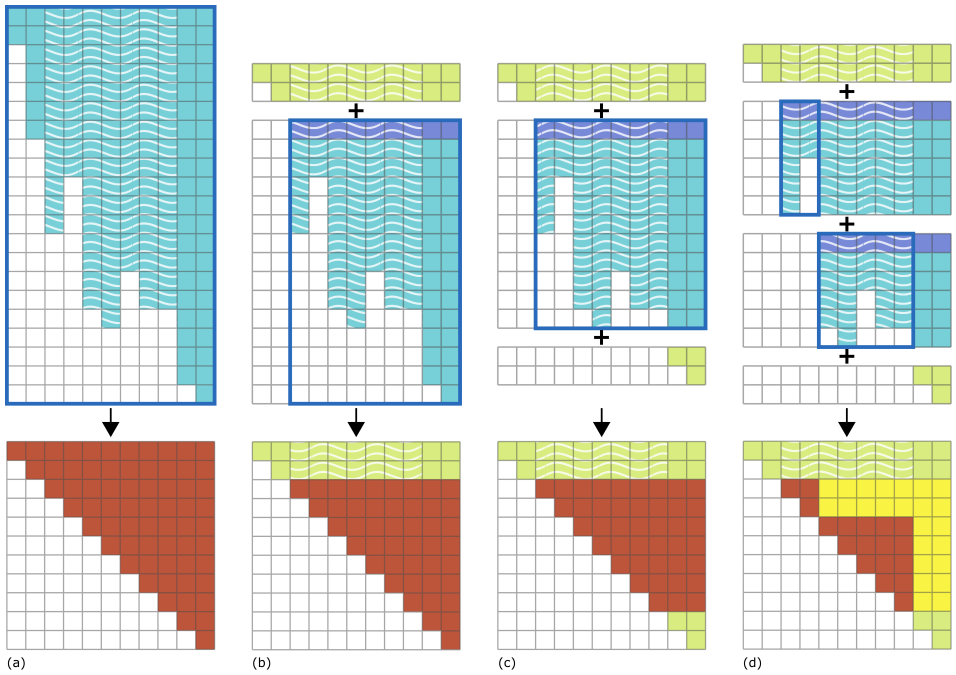

Figure 4: Modification via re-factorization. Blue borders mark the re-factorized sub-matrices; teal-colored entries represent entries of $\boldsymbol{A}$ (i.e., factors); green represents entries of $\boldsymbol{R}$; purple represents marginal factors; red represents entries calculated from re-factorization of $\boldsymbol{A}$; yellow represent entries calculated by applying a pre-calculated transformation; white overlay indicates column permutation. (a) Naive/complete re-factorization. (b) iSAM2-style incremental factorization, starting from the first permuted variable. (c) Utilizing the local effect of variable reordering. (d) Optimized re-factorization, as described in Algorithm 2; utilizing block independence and efficient block modification; the highlighted blocks can be factorized in parallel.

Furthermore, we can also utilize the concept of block independence, and divide rows $\mathtt{I}_{\boldsymbol{AP}}(j_{\text{first}}, j_{\text{last}})$ of $\boldsymbol{AP}$ into smaller independent row blocks, which can be factorized in parallel – just like we considered in direct modification. Here, though, to every row block we shall add its relevant marginal factors (which were pre-cached). Finally, in each of these row blocks, we can separate the calculation of the orthogonal transformation from its application, as previously suggested. For a block of rows of $\boldsymbol{AP}$ (with the marginal factors), representing l rows of $\boldsymbol{R_p}$, it is sufficient to examine its first l columns, in order to calculate the desired transformation. Only then we shall apply it to the remaining columns of that row block. We note, though, that in this case, this sub-block is not necessarily square, as the number of rows to re-factorize may be greater than the rank of the block. Application of these two concepts is visualized in Fig. 4d. Also, we recall that this optimized modification method is summarized in Algorithm 2.

V. Experimental Results

We would now like to demonstrate the advantages of our optimized modification technique, provided in Algorithm 2. As explained, variable reordering is highly relevant when performing online sequential estimation tasks. Accordingly, we solved a SLAM problem, in order to estimate the trajectory of a ground robot navigating in an unknown environment, in a highly realistic Gazebo simulation (for more details about the simulation, see [18]). At the end of the trajectory, the state vector contained nearly 400 (three-dimensional) poses. We then extracted the coefficient matrix $\boldsymbol{A}$, and the square root matrix $\boldsymbol{R} = \mathtt{chol}\{\boldsymbol{A}^T\boldsymbol{A}\}$, to be used in this demonstration; the sparsity pattern of $\boldsymbol{A}$ can be viewd in Fig. 5a. We examined three variants of direct modification: naive (i.e., $\mathtt{qr}\{\boldsymbol{RP}\}$); our optimized algorithm; and our optimized algorithm, while allowing parallel modification of independent blocks. We also examined four variants of modification via re-factorization (visible in Fig. 4): naive, i.e., complete re-factorization; incremental re-factorization, matching iSAM2; and our optimized algorithm, with and without parallelization. For fairness, our implementation[1] of all variants relied on the standard `qr` function in MATLAB. To attain unbiased results, we generated 100 random permutations, to be used for comparison. In order to still keep the number of blocks consistent, when generating each permutation, we first randomly selected two blocks of variables, each with a random number of poses, and within each of those, generated a random permutation of the poses.

The average results of applying these permutations, for each of the compared algorithms, are summarized in Table I; the table presents the execution time, and the number of NZs (NNZ), indicating the added fill-in due to numerical errors. The sparsity pattern of the modified factor $\boldsymbol{R_p}$ (considering a typical permutation) can be seen in Fig. 5b and Fig. 5c. Several observations can be made. First, the NNZ within each algorithm category is consistent, though direct modification generally yields additional fill-in in comparison to re-factorization. Nonetheless, in terms of execution time, direct optimization beats re-factorization by a large margin. As we expected, even without parallelization, in each category, our

[1]Implementation of the presented algorithms is available at `www.khen.io`

Table I: Comparison of modification algorithms.

Category	*Algorithm*	*Runtime (ms)*	*NNZ*
Direct	Naive	87	390k
	Optimized	13	392k
	Optimized (parallel)	11	392k
Re-factor	Naive	336	304k
	Incremental (iSAM2)	229	304k
	Optimized	83	307k
	Optimized (parallel)	70	307k

optimized algorithm achieves significantly better performance in comparison to the baseline methods. Allowing parallel computing can obviously further improve the performance, in direct relation to the number of concurrently modified blocks.

VI. Conclusion

We addressed the problem of modifying the upper triangular matrix $\boldsymbol{R}$, given by factorizing the coefficient matrix $\boldsymbol{A}$ of a linear(ized) least squares system, to convey reordering of the variables. We identified three main contributions, to allow efficient computation. First, variable reordering has only a local effect on the factor; meaning, $\boldsymbol{R}$ is affected only between its rows which correspond to the first and last permuted variables. Second, this row block can be divided into independent sub-blocks, which can be modified independently, and even in parallel. Third, to modify each of these blocks, it is sufficient to examine only the square portion of it, around the matrix diagonal, in order to derive the transformation required for the modification. We utilized these conclusions in a novel modification algorithm, and examined two variants of it: directly modifying $\boldsymbol{R}$, without re-accessing $\boldsymbol{A}$; and combining (partial) re-factorization of $\boldsymbol{A}$. As our simulation proves, the first enjoys lower computation time, while the latter enjoys higher accuracy. Nonetheless, considering either variant, our optimized algorithm significantly improves the performance over state-of-the-art. Overall, as the two variants show superiority in different criteria, selecting the one to use in a matter of preference. If execution time holds higher importance over reducing fill-in, one should use direct modification; if the opposite is true, one should use re-factorization. Though, we remind again that optimized re-factorization relies on marginal factor caching, which may not be available.

We can consider a "hybrid" modification, which enjoys the benefits of both methods. Modern sparse `qr` implementations often rely on a precursory symbolic factorization process. These start by calculating the sparsity pattern of $\boldsymbol{R}$, and identify the entries which are zero "by definition", and those which "need to be computed". They then calculate the numerical values only for the relevant entries. Since the rows of $\boldsymbol{R}$ are denser than those of $\boldsymbol{A}$, the sparsity pattern we can derive in direct modification is inferior to the one we derive in re-factorization; this causes more entries to suffer from numerical errors. Thus, we may begin by performing symbolic re-factorization, to derive the sparsity pattern of the modified factor, and then calculate the relevant numerical values via direct modification. To further generalize and extend the applicability of our ideas, we are also interested in analyzing them in the context of graphical models, such as the factor graph, or the Bayes tree (which underlies the iSAM2 algorithm). Formulation of these ideas is left for future work.

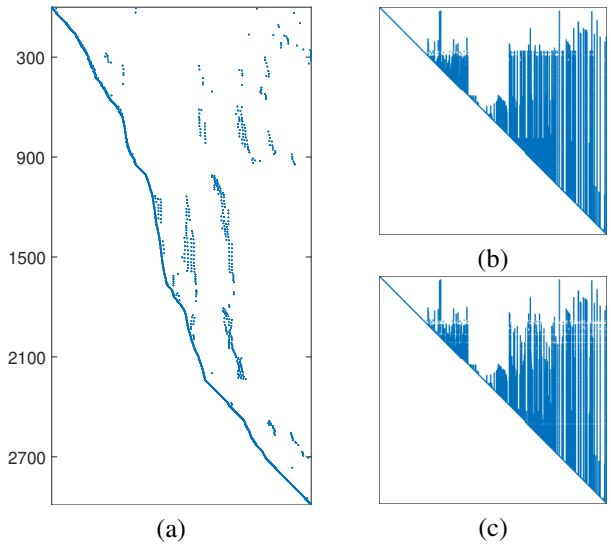

Figure 5: The sparsity pattern of $\boldsymbol{A}$ (a), and the modified factor $\boldsymbol{R_p}$, calculated using the two variants our optimized modification algorithm: direct modification (b), and re-factorization (c).

References

[1] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT press, Cambridge, MA, 2005.

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

[3] G. Golub and C. V. Loan, *Matrix Computations*, 3rd ed. Baltimore: Johns Hopkins University Press, 1996.

[4] T. Davis, *Direct Methods for Sparse Linear Systems*, ser. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, 2006.

[5] M. Yannakakis, "Computing the minimum fill-in is NP-complete," *SIAM J. Algebraic Discrete Methods*, vol. 2, 1981.

[6] T. Davis, J. Gilbert, S. Larimore, and E. Ng, "A column approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 353–376, 2004.

[7] P. Agarwal and E. Olson, "Variable reordering strategies for slam," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*. IEEE, 2012, pp. 3844–3850.

[8] K. Elimelech and V. Indelman, "Introducing pivot: Predictive incremental variable ordering tactic for efficient belief space planning," in *Proc. of the Intl. Symp. of Robotics Research (ISRR)*, October 2019.

[9] P. Gill, G. Golub, W. Murray, and M. Saunders, "Methods for modifying matrix factorizations," *Mathematics and Computation*, vol. 28, no. 126, pp. 505–535, 1974.

[10] T. Davis and W. Hager, "Modifying a sparse Cholesky factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 3, pp. 606–627, 1996.

[11] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate," *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, pp. 1–14, 2008.

[12] S. Touchette, W. Gueaieb, and E. Lanteigne, "Efficient cholesky factor recovery for column reordering in simultaneous localisation and mapping," *J. of Intelligent and Robotic Systems*, vol. 84, no. 1-4, pp. 859–875, 2016.

[13] M. Kaess, A. Ranganathan, and F. Dellaert, "iSAM: Incremental smoothing and mapping," *IEEE Trans. Robotics*, vol. 24, no. 6, pp. 1365–1378, Dec 2008.

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

[15] V. Ila, L. Polok, M. Solony, and P. Svoboda, "SLAM++ - a highly efficient and temporally scalable incremental slam framework," *Intl. J. of Robotics Research*, vol. 36, no. 2, pp. 210–230, 2017.

[16] F. Dellaert, A. Kipp, and P. Krauthausen, "A multifrontal QR factorization approach to distributed inference applied to multi-robot localization and mapping," in *Proc. 22^{nd} AAAI National Conference on AI*, Pittsburgh, PA, 2005, pp. 1261–1266.

[17] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Multifrontal qr factorization for multicore architectures over runtime systems," in *European Conference on Parallel Processing*. Springer, 2013, pp. 521–532.

[18] K. Elimelech and V. Indelman, "Efficient decision making and belief space planning using sparse approximations," *arXiv preprint arXiv:1909.00885*, 2018.