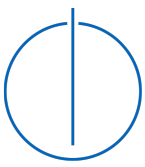
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Factor Grouping for Efficient Bundle Adjustment

Tin Chon Chan



DEPARTMENT OF INFORMATICS

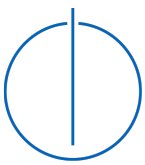
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Factor Grouping for Efficient Bundle Adjustment

Effizienter Bündelausgleich mittels Faktorgruppierung

Author:	Tin Chon Chan
Supervisor:	Prof. Dr. Daniel Cremers
Advisor:	Nikolaus Demmel, Simon Weber
Submission Date:	15.04.2022



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.04.2022 Tin Chon Chan

Acknowledgments

I would like to express my appreciation to those who have supported and motivated me during my study. First, I would like to thank my supervisor Prof. Dr. Daniel Cremers for introducing me to the field of computer vision through his excellent lectures.

Second, I would like to express my sincere gratitude to my advisors Nikolaus Demmel and Simon Weber, who made this work possible with their patience guidance and comprehensive expertise in this topic. They are always willing to provide me valuable insight into the mathematical and programming details.

Lastly, I am deeply grateful to my family, friends, and my girlfriend Ye Liu for their unhesitating encouragement and support throughout my study.

Abstract

Bundle adjustment (BA) is a method of refining the estimation of camera poses and parameters, and the 3D points extracted from the captured images. It is a key component of 3D vision applications, such as Structure-from-Motion and Simultaneous Localization and Mapping. However, BA can be slow to solve if the optimization routine is not specially developed with the special structure of BA in mind. The efficiency of a BA solver is particularly crucial for solving large-scale BA problems, which can result in substantial runtime differences.

In this work, we study the efficiency and performance of different BA solvers. We examine the performance of applying the Schur Complement (SC) trick in the explicit and implicit variant, as well as the recently proposed square root formulation. Then, we revisit the concept of grouping 3D landmarks into factor groups and reimplement this factor SC solver as a mix between implicit and explicit SC. Moreover, we propose a new implicit square root solver and combine it with the factor grouping scheme. Lastly, we present a novel solver which directly approximates the inverse of the Schur complement as a power series instead of using preconditioned conjugate gradient.

All solvers are implemented in the same C++ framework, where operations are efficiently parallelized on multiple CPU cores and supported by SIMD vectorization. Our solvers are extensively evaluated on multiple real world datasets, which include a wide diversity of problems. We investigate and compare the characteristics of the solvers from different perspectives supported by multiple visualizations, and analyse important implementation details such as the memory access pattern, multiplication order, and recomputing values versus retrieving them from cache.

Based on our findings we make a recommendation on the best solver choice depending on the scenario at hand. The implicit SC solver is unmatched for accurately solving large-scale and dense problems. For small- and medium-scale problems the factor SC solver provides good all-round performance. If computations have to be done only in single-precision, the implicit square root solver can still offer excellent accuracy, even if on large dense problems it requires more memory and runtime. Power BA is recommended for those that can perform optimization in double precision and value speed over accuracy. It is significantly faster in the initial cost reduction than other solvers but can have lower final accuracy.

Contents

1 Introduction

Motivation With the rising performance of modern computer hardware, many 3D vision applications are viable to run on a ubiquitous computer. Subsequently, autonomous driving may become reality thus attracting a strong interest from the research communities. Two of the most essential components of autonomous driving are Simultaneous localization and mapping (SLAM) and Structure from Motion (SfM).

SLAM is a method for simultaneously mapping the surrounding environment and tracking the pose of a vehicle. It is critical for autonomous driving since it enables a vehicle to navigate through an unknown environment. However, using SLAM alone for navigation may create small errors that accumulates over time, or erroneous navigation may occur in extreme weather circumstances, which may result in serious catastrophe.

SfM is a method for reconstructing the three-dimensional structure of a scene from a collection of unordered images. In comparison to SLAM, it places a higher emphasis on the reconstruction accuracy than the real-time capability. SfM can be used to create High Definition (HD) maps that comprise detailed 3D road information, such as pedestrian crossings, traffic lights, and lane placement. HD maps enable a more precise navigation result and make it easy to identify moving objects.

Bundle Adjustment (BA) plays an important role in both SLAM and SfM. It is a non-linear optimization process that refines the initial estimation of camera poses and 3D landmarks jointly. Since using BA substantially improves the accuracy of navigation and 3D reconstruction, BA is often a core component of the majority of state-of-the-art 3D vision systems. However, the runtime of BA can be a bottleneck of these systems, especially for large-scale SfM tasks such as creating HD maps, where the reconstruction can be the size of a city.

Numerous efforts have been made to improve the runtime of BA. Levenberg-Marquardt is an algorithm that can convert a BA problem into a linearized problem, and then it approximates the solution of the original problem iteratively. It has proved to be a very effective strategy for solving BA problems. The Schur complement (SC) trick is a commonly used technique for significantly reducing the dimension of the linearized problem by leveraging the special structure of BA optimization. It can improve the speed of solving BA problems, and make it tractable to solve on a single computer. The square root formulation [Dem+21] is an alternative technique for reducing the dimension. It makes use of nullspace marginalization to eliminate the landmark variables

from the optimization by QR decomposition.

There are various ways of solving the reduced linearized BA problem, two of the most frequently used are Cholesky factorization and preconditioned conjugate gradient (PCG). Cholesky factorization decomposes the linear system, and then directly solves the system by forward and backward substitution. It is typically adopted to solve small-scale problems due to its cubic time complexity and quadratic space complexity in the number of cameras. On the other hand, PCG approximates the solution of the linear system iteratively. It solves the system by performing vector-matrix multiplication and has a lower time complexity than factorization methods. In this work, PCG is adopted to solve BA problems since it scales better to large-scale problems.

By applying the Schur complement trick to the linearized BA system, we can obtain the reduced system, which is known as reduced camera system (RCS). The matrix representation of RCS is called reduced camera matrix, which is composed on multiple matrices. Since solving via PCG only requires the vector-matrix multiplication with the reduce camera matrix, we can either construct the reduced camera matrix explicitly or perform multiplications implicitly with the individual matrices sequentially. As demonstrated in [Aga+10], multiplication with the implicit representation of reduced camera matrix is faster in practice.

Factor grouping [Car+14] has been proposed to accelerate the optimization by mixing the explicit and implicit representation. By representing a BA problem on a factor graph, it demonstrates that the landmarks can be grouped into multiple factor groups. Since each factor group can be represented as a RCS, factor grouping focuses on selecting the optimal representation between explicit or implicit representation based on the multiplication time complexity.

In this work, we study the performance of different BA optimization solvers on problems with different scales. Our contributions are:

- In addition to the two solvers (explicit SC and $\sqrt{BA}$ solver) implemented in [Dem+21], we develop five new solvers written in C++ within the same framework. We reimplement two solvers that use implicit RCS representation and apply factor grouping scheme to solve RCS, respectively. After that, we develop and implement two new square root solvers that, respectively, marginalize landmarks on-the-fly and apply factor grouping to square root formulation. Additionally, we present and implement a novel solver that solves the RCS by directly approximating the inverse of reduce camera matrix by power series.

- The implementation of our solvers are highly optimized for efficiency. The operations are parallelized on multiple CPU cores and vectorized with SIMD instructions.

- We perform extensive evaluation of the seven solvers on three datasets with distinct characteristics: BAL [Aga+10], 1DSfM [WS14], and MCBA [Wu+11]. In addition, we also compare the performance of our explicit and implicit SC solvers with Ceres implementation. We illustrate and analyze our experimental results from different perspectives through multiple visualizations.
- In addition to the overall performance, we investigate the specific aspects of our implementations, including time complexity, memory access pattern, and numerical properties.

Related Work The mathematical background of BA is already well established. The special structure of BA problem and the details of commonly used optimization algorithms including LM, PCG, and Schur complement trick, can be found in [Tri+99]. Multiple open-source libraries for solving BA problems written in C++ are widely accessible, in which SBA package [LA09], g2o framework [Gri+11], and Ceres solver [AMT22] are the most popular ones.

SBA discusses the efficient implementation design of a BA solver, and demonstrates the superior performance of custom BA solver against a general purpose solver when solving BA. g2o presents a graph-based non-linear optimization framework designed for BA. Ceres is a well-written optimization framework which can be used for solving both generic and BA problems, which is a popular choice in the computer vision and robotics communities.

Many works in the field of BA have been proposed to accelerate BA. [Aga+10] proposes a custom preconditioner for PCG to solve RCS. They build a dataset from internet images and implement different solvers to extensively evaluate their proposed method. [Car+14] expresses a BA problem on factor group and tries to divide landmarks into groups, then it exploits the time complexity of multiplication to reduce the multiplication runtime. [Wu+11] investigates an efficient implementation to solve RCS via PCG by accelerating operations on multiple CPU cores or GPU. Similarly, they generate a large-scale dataset for evaluating their implementation. [Ren+21] explores an efficient GPU implementation which can be parallelized on multiple GPUs.

Outline The content of this thesis is organized as following: In Chapter 2, we give the concrete definition of BA, and introduce corresponding mathematical background and notion. In Chapter 3, the mathematical and implementation details of our solvers are explained. In Chapter 4, the datasets which are used to evaluate our solvers are introduced. After that, we study the performance of the solvers through analysis and visualization of the experimental results. In Chapter 5, we summarize our findings and recommend the solver to use given certain situations.

2 Background

Structure from Motion is the task of reconstructing the 3D geometry of a particular scene from a given set of images. In our experiments we perform this task for unordered internet image collections. Therefore, we assume that each image is captured by a separate camera and hence for each camera we estimate a different set of not only position and orientation, but also intrinsic parameters.

Given n_p images and N_r observations of all landmarks in all images, in which each observation is a 2D pixel coordinate $\mathbf{u} \in \mathbb{R}^2$, an initial estimation of n_p sets of camera parameters and n_l landmarks can be extracted using a structure from motion pipeline. The 3D structure is represented as a set of sparse landmarks $\{\mathbf{x}_i\}_{i=1}^{n_l} \subset \mathbb{R}^3$. A landmark is a distinct point in the real world observed by multiple cameras, it can be computed by projecting the 2D pixel coordinate of its observations to 3D space, and then a estimation of its 3D coordinate is computed by triangulation. Consequently, landmarks are closely related to the camera parameters by the projection, thus they are also needed to be refined together with the landmarks.

In this work, the parameters of a camera are the rotation matrix $\mathbf{R} \in SO(3)$, translation $\mathbf{t} \in \mathbb{R}^3$, focal length $f \in \mathbb{R}_{>0}$ and distortion parameters $\mathbf{k} \in \mathbb{R}^2$. Given the parameters, a landmark i can be projected to a 2D pixel coordinate, it then can be used to compute the reprojection residual using the observation $\mathbf{u}_{ij}$ of the landmark captured by camera j. At the end, the 3D structure and the camera parameters can be refined by minimizing reprojection residuals using bundle adjustment.

In the following sections, rigid body motion which can represent camera poses is first presented. Then Lie Algebra is introduced to allow a simpler representation of rigid body motion. After that, camera projection model, reprojection residual and bundle adjustment are introduced in sequence. Finally, optimization techniques for bundle adjustment are presented.

2.1 Problem formulation

2.1.1 Rigid Body Motion

A rigid body motion is defined as a family of maps:

$$g_t : \mathbb{R}^3 \to \mathbb{R}^3;\ \mathbf{x} \to g_t(\mathbf{x}),\ t \in [0, T]\,, \tag{2.1}$$

which preserves the norm and cross products of any two vectors:

$$||g(\mathbf{u})|| = ||\mathbf{u}||, \forall \mathbf{u} \in \mathbf{R}^3\,, \tag{2.2}$$

$$g(\mathbf{u}) \times g(\mathbf{v}) = g(\mathbf{u} \times \mathbf{v}), \forall \mathbf{u}, \mathbf{v} \in \mathbb{R}^3\,. \tag{2.3}$$

It implies g_t preserves the length and orientation, the motion g_t of a rigid body can be represented by the motion of a Cartesian coordinate frame attached to the rigid body.

The special orthogonal group $SO(3)$, i.e. the set of 3D rotation matrices, can be used for representing rotations in 3D:

$$SO(3) = \{\mathbf{R}^{3\times3} |\ \mathbf{R}^\top\mathbf{R} = \mathbf{I},\ \det(\mathbf{R}) = 1\}\,. \tag{2.4}$$

Then, the rigid body motion g_t can be written as:

$$g_t(\mathbf{x}) = \mathbf{R}\mathbf{x} + \mathbf{t},\ \mathbf{R} \in SO(3), \mathbf{t} \in \mathbf{R}^3\,. \tag{2.5}$$

By specifying the rotation matrix $\mathbf{R}$ and translation vector $\mathbf{t}$, a rigid body motion is uniquely defined.

More conveniently, the space of rigid body motions in $\mathbb{R}^3$ can be given by the special Euclidean group $SE(3)$, which combines rotation and translation:

$$SE(3) = \left\{ \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \middle|\ \mathbf{R} \in SO(3),\ \mathbf{t} \in \mathbb{R}^3 \right\}\,. \tag{2.6}$$

Recognizing rotation matrix $\mathbf{R}$ as the camera orientation and translation vector $\mathbf{t}$ as the camera position, a camera pose can be written as:

$$T = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{pmatrix} \in SE(3)\,. \tag{2.7}$$

It is a transformation matrix which can map a landmark from world to camera coordinate system (the origin of which is attached to the center of camera).

Skew Symmetric Matrices

A 3D skew symmetric matrix must satisfy:

$$\mathbf{A}^\top = -\mathbf{A},\ \mathbf{A} \in \mathbb{R}^{3\times3}\,. \tag{2.8}$$

Such skew symmetric matrices can be written as:

$$\hat{\mathbf{u}} = \begin{pmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{pmatrix},\ \mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}\,. \tag{2.9}$$

A skew symmetric matrix can be uniquely identified with a vector $\mathbf{u} \in \mathbb{R}^3$. The operator $\cdot^\wedge$ defines an isomorphism between $\mathbb{R}^3$ and the space $\mathfrak{so}(3) = \{\hat{\mathbf{u}} | \ \mathbf{u} \in \mathbb{R}^3\}$. The inverse is defined as $\cdot^\vee : \mathfrak{so}(3) \to \mathbb{R}^3$.

Skew symmetric matrices can model the cross product on $\mathbb{R}^3$:

$$\mathbf{u} \times \mathbf{v} = \hat{\mathbf{u}}\mathbf{v} \,. \tag{2.10}$$

Lie Algebra

A matrix $\mathbf{R}^{3\times3}$ has 9 DoF (Degree of Freedom), but rotations $\mathbf{R} \in SO(3)$ can be represented with 3DoF. By establishing the mapping between $SO(3)$ and $\mathfrak{so}(3)$, we can represent rotation matrices with 3DoF, therefore $\mathfrak{so}(3)$ is called the Lie algebra of Lie group $SO(3)$. Otherwise if the rotation matrix R is optimized directly, one needs to setup a constrained optimization, or project R back to the space of $SO(3)$ at every iteration.

In this thesis, we do not consider rotations independently, but always as part of a camera pose. Therefore, we skip the discussion of the Lie Group of rotations $SO(3)$ and its Lie algebra $\mathfrak{so}(3)$ and directly move on to the Lie Group of rigid body motions $SE(3)$ and its Lie algebra $\mathfrak{se}(3)$, which it is more convenient to represent camera poses ($\mathbf{R} \in \mathbb{R}^{3\times3}$ and $\mathbf{t} \in \mathbb{R}^3$) with $\mathfrak{se}(3)$ for the same reason as $\mathfrak{so}(3)$. The interested readers are referred to [Ead13] for a more detail introduction.

The Lie algebra of $SE(3)$, $\mathfrak{se}(3)$ is defined as:

$$\mathfrak{se}(3) = \left\{ \hat{\xi} = \begin{pmatrix} \hat{\mathbf{w}} & \mathbf{v} \\ \mathbf{0} & 0 \end{pmatrix} \middle| \ \hat{\mathbf{w}} \in \mathfrak{so}(3),\, \mathbf{v} \in \mathbb{R}^3 \right\} \subset \mathbb{R}^{4\times4} \,. \tag{2.11}$$

The operators $\cdot^\wedge$ and $\cdot^\vee$ to convert between a twist $\hat{\xi}$ and its twist coordinates $\xi \in \mathbb{R}^6$ are defined as:

$$\begin{aligned} \hat{\xi} = \begin{pmatrix} \mathbf{v} \\ \mathbf{w} \end{pmatrix}^\wedge &= \begin{pmatrix} \hat{\mathbf{w}} & \mathbf{v} \\ \mathbf{0} & 0 \end{pmatrix} \in \mathfrak{se}(3) \,, \\ \begin{pmatrix} \hat{\mathbf{w}} & \mathbf{v} \\ \mathbf{0} & 0 \end{pmatrix}^\vee &= \begin{pmatrix} \mathbf{v} \\ \mathbf{w} \end{pmatrix} \in \mathbb{R}^6 \,, \end{aligned} \tag{2.12}$$

in which the twist coordinate ξ comprises of a translational component $\mathbf{v} \in \mathbb{R}^3$ and rotational component $\mathbf{w} \in \mathbb{R}^3$.

The exponential map $\exp : se(3) \to SE(3)$ can map $se(3)$ to $SE(3)$ in closed-form:

$$\begin{aligned} \exp(\hat{\xi}) &= \begin{pmatrix} \exp(\hat{\mathbf{w}}) & \mathbf{J}\mathbf{v} \\ \mathbf{0} & 1 \end{pmatrix} \,, \\ \mathbf{J} &= \mathbf{I} + \frac{1-\cos(\theta)}{\theta^2}\hat{\mathbf{w}} + \frac{\theta - \sin(\theta)}{\theta^3}\hat{\mathbf{w}}^2 \,. \end{aligned} \tag{2.13}$$

The inverse can be computed by the logarithmic map in closed-form:

$$\begin{aligned}\begin{pmatrix}\mathbf{v}\\ \mathbf{w}\end{pmatrix} &= \log\begin{pmatrix}\mathbf{R} & \mathbf{t}\\ \mathbf{0} & 1\end{pmatrix}^{\vee},\\ \mathbf{w} &= \log(\mathbf{R}),\\ \mathbf{v} &= \mathbf{J}^{-1}\mathbf{t},\\ \mathbf{J}^{-1} &= \mathbf{I} - \frac{1}{2}\hat{\mathbf{w}} + \left(\frac{1}{\theta^2} - \frac{1+\cos(\theta)}{2\theta\sin(\theta)}\right)\hat{\mathbf{w}}^2,\end{aligned} \tag{2.14}$$

where $\theta = \|w\|$.

2.1.2 Camera Projection Model

Since we have a single set of intrinsic parameters per image, it is favorable to use a minimal representation to reduce the number of parameters needed to be optimized. In this thesis, we use a simple pinhole camera model with a single parameter focal length, and a two-parameter radical distortion.

Assuming the pixels are perfect unit squares, a landmark $\mathbf{x} = (x\ y\ z)^\top$ can be projected to the image coordinate using its camera parameters:

$$\begin{pmatrix}x'\\ y'\\ z'\\ 1\end{pmatrix} = \underbrace{\begin{pmatrix}\mathbf{R} & \mathbf{t}\\ \mathbf{0} & 1\end{pmatrix}}_{\mathbf{T}}\begin{pmatrix}x\\ y\\ z\\ 1\end{pmatrix}. \tag{2.15}$$

The landmark $\mathbf{x}$ is first augmented with 1 to form a homogeneous coordinate, then transformed from world to camera coordinate system by multiplying with camera transformation matrix $\mathbf{T} \in SE(3)$.

Then we can compute the image coordinate $\mathbf{x}' \in \mathbb{R}^2$:

$$\mathbf{x}' = \begin{pmatrix}\frac{x'}{z'}\\ \frac{y'}{z'}\end{pmatrix}. \tag{2.16}$$

Images captured with average cameras are often distorted, especially at shorter focal lengths. Such kind of radial distortion can be modeled and used in reprojecting landmarks for computing reprojection residual. In our experiments, the image coordinate $\mathbf{x}'$ is distorted using a simple two-parameter model:

$$\mathbf{k} = \begin{pmatrix}k_1\\ k_2\end{pmatrix}, \tag{2.17}$$

$$\mathbf{x}'_d = (1 + k_1\|\mathbf{x}'\|^2 + k_2\|\mathbf{x}'\|^4)\mathbf{x}'. \tag{2.18}$$

Finally, the distorted image coordinate $\mathbf{x}'_d$ is projected to the pixel coordinate $\mathbf{u}'$ by multiplying with a focal length f:

$$\mathbf{u}' = f\mathbf{x}'_d \,. \tag{2.19}$$

For simplicity, we define a projection function $\pi : \mathbb{R}^3 \to \mathbb{R}^2$, which projects a landmark $\mathbf{x}$ to the 2D pixel coordinate $\mathbf{u}'$ using a set of camera parameters as described above:

$$\mathbf{u}' = \pi(\mathbf{R}\mathbf{x} + \mathbf{t};\ f\,, k)\,. \tag{2.20}$$

2.1.3 Bundle adjustment

Bundle adjustment is the process of refining initial estimates of 3D landmark positions and camera poses. This is a central component of a structure from motion pipeline and the main topic of this thesis. In bundle adjustment we minimize the reprojection error over all landmark observations, that is the difference between observed and reprojected feature position in image space. For a 3D landmark i which was detected in image j at keypoint position $\mathbf{u}_{ij} \in \mathbb{R}^2$, we define the reprojection error residual as:

$$\mathbf{r}_{ij} = \mathbf{u}_{ij} - \pi(\mathbf{R}_i\mathbf{x}_j + \mathbf{t}_i;\ f_i,\ \mathbf{k}_i)\,, \tag{2.21}$$

where it computes the difference between the given pixel coordinate in the image and the reprojected coordinate.

We define O_i as the set of cameras observing the landmark i, $\mathbb{P} = SE(3) \times \mathbb{R}_{>0} \times \mathbb{R}^2$ as the set of camera states, and $\mathbb{L} = \mathbb{R}^3$ as the set of landmark states. Then, the objective function of the bundle adjustment can be written as,

$$E(\mathbf{x}) = \sum_{i=1}^{n_l} \sum_{j \in O_i} \frac{1}{2} \|\mathbf{r}_{ij}\|^2 = \frac{1}{2}\|\mathbf{r}\|^2\,, \tag{2.22}$$

where $\mathbf{x} \in \mathbb{X} = \mathbb{P}^{n_p} \times \mathbb{L}^{n_l}$ is optimization state of all the cameras and landmarks in the problem. The optimization state is updated using the operator $\boxplus$, which is defined as:

$$\boxplus : \mathbb{X} \times \mathbb{R}^6 \to \mathbb{X}\,, \tag{2.23}$$

$$\mathbf{x} \boxplus \Delta\mathbf{x} = \exp(\Delta\mathbf{x})\mathbf{x}\,. \tag{2.24}$$

For convenience, the individual residuals $\mathbf{r}_{ij}$ will be stacked as a vector $\mathbf{r} \in \mathbb{R}^{2N_r}$.

2.2 Optimization

2.2.1 Levenberg-Marquardt

Levenberg-Marquardt (LM) is an algorithm for solving non-linear least square problems, it has proven to be one of the most effective algorithm to solve bundle adjustment. Following Section 2.1.3, the non-linear objective function of bundle adjustment (2.22) is minimized:

$$\min_{\mathbf{x}\in\mathbb{X}} E(\mathbf{x})\,, \tag{2.25}$$

$$E(\mathbf{x}) = \frac{1}{2}\|\mathbf{r}\|^2\,. \tag{2.26}$$

Similar to the Gauss-Newton algorithm, the LM algorithm minimizes the objective function by solving a sequence of approximations of the original problem. At each LM iteration, the residual is approximated by a first order Taylor expansion around the current value of $\mathbf{x}$:

$$E(\mathbf{x}\boxplus\Delta\mathbf{x}) \approx \frac{1}{2}\|\mathbf{r}+\mathbf{J}\Delta\mathbf{x}\|^2\,, \tag{2.27}$$

where $\mathbf{J}\in\mathbb{R}^{2N_r\times(9n_p+3n_l)}$ is the Jacobian matrix.

Instead of solving for $\mathbf{x}$ directly, we parameterize the linearized residual by a perturbation $\Delta\mathbf{x}\in\mathbb{R}^{9n_p+3n_l}$ in the tangent space of the current estimate $\mathbf{x}$. Then, the minimizing problem in every iteration becomes:

$$\min_{\Delta\mathbf{x}} E_{\text{lin}}(\Delta\mathbf{x})\,, \tag{2.28}$$

$$E_{\text{lin}}(\Delta\mathbf{x}) = \frac{1}{2}\|\mathbf{r}+\mathbf{J}\Delta\mathbf{x}\|^2\,. \tag{2.29}$$

We can solve this linear least squares problem with the normal equations:

$$\begin{aligned}\frac{\partial}{\partial\Delta\mathbf{x}}\frac{1}{2}\|\mathbf{r}+\mathbf{J}\Delta\mathbf{x}\|^2 &= \frac{\partial}{\partial\Delta\mathbf{x}}(\frac{1}{2}\mathbf{r}^\top\mathbf{r}+\mathbf{r}^\top\mathbf{J}\Delta\mathbf{x}+\frac{1}{2}\Delta\mathbf{x}^\top\mathbf{J}^\top\mathbf{J}\Delta\mathbf{x})\\ &= \mathbf{J}^\top\mathbf{r}+\mathbf{J}^\top\mathbf{J}\Delta\mathbf{x}\stackrel{!}{=}\mathbf{0}\\ \Rightarrow \mathbf{J}^\top\mathbf{J}(-\Delta\mathbf{x}) &= \mathbf{J}^\top\mathbf{r}\,.\end{aligned} \tag{2.30}$$

At the end of each LM iteration, $\mathbf{x}$ is updated as:

$$\mathbf{x}\leftarrow\mathbf{x}\boxplus\Delta\mathbf{x}\,. \tag{2.31}$$

To improve the convergence rate, LM includes a damping parameter that controls the step size in the objective function:

$$\min_{\Delta\mathbf{x}} \frac{1}{2}\|\mathbf{r}+\mathbf{J}\Delta\mathbf{x}\|^2+\frac{1}{2}\lambda\|D\Delta\mathbf{x}\|^2\,, \tag{2.32}$$

where D is a non-negative diagonal matrix and $\lambda \in \mathbb{R}_{\geq 0}$ is a parameter controlling the strength of regularization. A popular choice of D is the square root of the diagonal of the matrix $\mathbf{J}^\top\mathbf{J}$, i.e. $\mathbf{D}^2 = \text{diag}(\mathbf{J}^\top\mathbf{J})$. Consequently, the step size is inversely proportional to λ.

Note that $\mathbf{H} = \mathbf{J}^\top\mathbf{J}$ is an approximation of the Hessian matrix of the nonlinear least squares energy E. With the damping parameter, the normal equations (2.30) become:

$$(\mathbf{H} + \lambda\mathbf{D}^2)(-\Delta\mathbf{x}) = \mathbf{J}^\top\mathbf{r}\,. \tag{2.33}$$

At each LM iteration, damping value λ will change based on whether the current update is accepted. If the update is accepted, $\mathbf{x}$ is updated as (2.31), and λ is decreased thus the cost of objective function (2.32) is also decreased; if the update is rejected, $\mathbf{x}$ will be unchanged, and λ is increased thus the cost of objective function (2.32) is also increased. The update decision is based on whether the current approximation of the objective function is good or poor. We skip the details of LM algorithm in this thesis, interested readers are referred to [Ran04].

The introduction of damping makes LM a combination of Gauss-Newton and gradient descent. When the current approximation of the objective function is good (λ is small), it behaves like Gauss-Newton, which has faster convergence rate; when the approximation is poor (λ is large), it behaves like gradient descent, which converge slower but assure to be converged. It is empirically proved to be very effective in solving bundle adjustment problems, therefore it is used in our work.

Huber Loss

To mitigate the negative effect of outliers, Huber loss is applied to the reprojection residuals. The Huber loss function is defined as:

$$L_\delta(\mathbf{r}) = \begin{cases} \frac{1}{2}\|\mathbf{r}\|^2 & \text{if} \|\mathbf{r}\| \leq \delta \\ \delta(\|\mathbf{r}\| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}, \tag{2.34}$$

where $\mathbf{r} \in \mathbb{R}^2$ is the reprojection residual of a single observation defined in (2.21).

The non-linear least square BA objective (2.22) is minimized using iteratively reweighted least squares with Huber loss. The normal equation (2.30) then can be written as:

$$\mathbf{J}^\top\mathbf{W}\mathbf{J}(-\Delta\mathbf{x}) = \mathbf{J}^\top\mathbf{W}\mathbf{r}\,, \tag{2.35}$$

where $\mathbf{W} \in \mathbb{R}^{(9n_p+3n_l)\times(9n_p+3n_l)}$ is a diagonal matrix, and the diagonal entries contains weights corresponding to the current residuals and Huber loss.

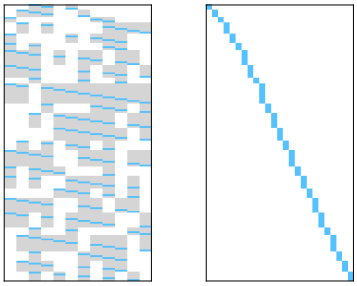


Figure 2.1: The rearranged Jacobian matrix $\mathbf{J} = \begin{pmatrix} \mathbf{J}_p & \mathbf{J}_l \end{pmatrix}$ [Dem+21]. The columns are rearranged, such that $\mathbf{J}_p \in \mathbb{R}^{2N_r \times 9n_p}$ only relates to cameras, and $\mathbf{J}_l \in \mathbb{R}^{2N_r \times 3n_l}$ relates to landmarks. Likewise, for the rows, the observations of each landmark are placed together, such that each landmark forms a non-zero block in $\mathbf{J}_l$.

Jacobian scaling

To mitigate numeric issues during optimization, we all normalized the columns of the Jacobian matrix $\mathbf{J}$ by:

$$\mathbf{j} \leftarrow \frac{\mathbf{j}}{\|\mathbf{j}\| + \epsilon} \text{ , for some small } \epsilon \text{ ,} \tag{2.36}$$

where $\mathbf{j} \in \mathbb{R}^{2N_r}$ is a column of the Jacobian matrix $\mathbf{J}$.

In our work, normalizing the Jacobian is applied after scaling with the Huber loss. We will omit both the Huber loss and scaling the Jacobian matrix in other sections for simplicity.

2.2.2 Schur Complement

Different from general optimization, bundle adjustment has a very special structure. Notably, the relationship of landmarks and camera parameters are reflected by looking at the Jacobian structure and Hessian structure.

For the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{2N_r \times (9n_p + 3n_l)}$, cameras contribute 9 columns ($\dim(\mathbb{P})$), landmarks contribute 3 columns ($\dim(\mathbb{L})$), and each observation contributes 2 rows. For each observation, only the position of the camera and landmark involved in the corresponding reprojection residual will be non-zero. Accordingly, the Jacobian can be rearranged as $\mathbf{J} = \begin{pmatrix} \mathbf{J}_p & \mathbf{J}_l \end{pmatrix}$ as shown in Figure 2.1.

With the rearranged Jacobian matrix, we can divide the Hessian matrix $\mathbf{H} \in$

$\mathbb{R}^{(9n_p+3n_l)\times(9n_p+3n_l)}$ in (2.33) into four sub-blocks:

$$\begin{aligned} \mathbf{H} &= \begin{pmatrix} \mathbf{H}_{pp} & \mathbf{H}_{pl} \\ \mathbf{H}_{lp} & \mathbf{H}_{ll} \end{pmatrix}, \\ \mathbf{H}_{pp} &= \mathbf{J}_p^\top \mathbf{J}_p, \\ \mathbf{H}_{ll} &= \mathbf{J}_l^\top \mathbf{J}_l, \\ \mathbf{H}_{pl} &= \mathbf{J}_p^\top \mathbf{J}_l = \mathbf{H}_{lp}^\top, \end{aligned} \tag{2.37}$$

where $\mathbf{H}_{pl} \in \mathbb{R}^{9n_p\times 3n_l}$ and $\mathbf{H}_{lp} \in \mathbb{R}^{3n_l\times 9n_p}$ are sparse block matrices, $\mathbf{H}_{pp} \in \mathbb{R}^{9n_p\times 9n_p}$ and $\mathbf{H}_{ll} \in \mathbb{R}^{3n_l\times 3n_l}$ are invertible block diagonal matrices with block size $\mathbb{R}^{9\times 9}$ and $\mathbb{R}^{3\times 3}$ respectively.

With the special structure of $\mathbf{H}$, we can rewrite the normal equation (2.33) (damping is omitted for simplicity) as a block-structured linear system:

$$\underbrace{\begin{pmatrix} \mathbf{H}_{pp} & \mathbf{H}_{pl} \\ \mathbf{H}_{lp} & \mathbf{H}_{ll} \end{pmatrix}}_{\mathbf{H}} \underbrace{\begin{pmatrix} -\Delta\mathbf{x}_p \\ -\Delta\mathbf{x}_l \end{pmatrix}}_{\Delta\mathbf{x}} = \underbrace{\begin{pmatrix} \mathbf{b}_p \\ \mathbf{b}_l \end{pmatrix}}_{\mathbf{Jr}}, \tag{2.38}$$

where $\Delta\mathbf{x}$ are $\mathbf{Jr}$ are both divided into camera related and landmark related parts:

$$\begin{aligned} \mathbf{b}_p &= \mathbf{J}_p^\top r, \\ \mathbf{b}_l &= \mathbf{J}_l^\top r. \end{aligned} \tag{2.39}$$

Given the special structure of $\mathbf{H}$, Schur complement trick can be used to define:

$$\begin{aligned} \widetilde{\mathbf{H}}_{pp} &= \mathbf{H}_{pp} - \mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp} \\ \widetilde{\mathbf{b}}_p &= \mathbf{b}_p - \mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{b}_l. \end{aligned} \tag{2.40}$$

Then (2.38) can be reduced to:

$$\widetilde{\mathbf{H}}_{pp}(-\Delta\mathbf{x}_p) = \widetilde{\mathbf{b}}_p. \tag{2.41}$$

$\widetilde{\mathbf{H}}_{pp}$ is known the Schur complement of $\mathbf{H}$ or Reduced Camera System (RCS).

For optimal landmark update $\Delta\mathbf{x}_l^*$, it can be solved in closed-form back substitution with the optimal camera parameters update $\Delta\mathbf{x}_p^*$:

$$-\Delta\mathbf{x}_l^* = \mathbf{H}_{ll}^{-1}(\mathbf{b}_l - \mathbf{H}_{lp}(-\Delta\mathbf{x}_p^*)). \tag{2.42}$$

In a typical bundle adjustment problem, the number of landmarks is always tremendously larger than the number of cameras. By using the Schur complement trick, landmarks are eliminated from the original system and only cameras remain. The

size of the system is significantly reduced from $(9n_p + 3n_l)^2$ to $(9n_p)^2$, Consequently making larger problems tractable to solve on a single computer.

Moreover, only the block diagonal matrices $\mathbf{H}_{pp}$ and $\mathbf{H}_{ll}$ are required to be inverted in the progress. Unlike inverting general matrices, block diagonal matrices can be inverted efficiently by inverting the small blocks independently instead of inverting the whole matrix.

In this thesis, we solve a bundle adjustment problem by optimizing the RCS (2.41) instead of the normal equations (2.33).

2.2.3 Preconditioned Conjugate Gradient

To solve the symmetric positive definite linear system (2.41), there are primarily two options: direct and iterative methods. Direct methods usually first factorize the linear system and solve it by back-substitution. They require space and time that can scale quadratically to the number of parameter and the number can be considerably large for bundle adjustment problems. Hence direct methods are only feasible for small to medium problems. In contrast, iterative methods solve the linear system by approximation. They need significantly less space for larger problems, hence they are an appropriate choice for solving medium to large bundle adjustment problems (typically 10^3 to 10^4 cameras or more).

Conjugate gradient is an iterative indirect method, it can solve a linear system by only requiring matrix-vector multiplications with $\mathbf{A}$:

$$\mathbf{A}\mathbf{x} = \mathbf{b}\,, \tag{2.43}$$

where $\mathbf{x} \in \mathbb{R}^n$ is an unknown solution of the system, $\mathbf{b} \in \mathbb{R}^n$ is a know vector, and $\mathbf{A} \in \mathbb{R}^{n\times n}$ is a known symmetric positive definite matrix.

However if $\mathbf{A}$ is poorly conditioned, it may result in very slow convergence rate. To improve the performance, the problem can be replaced with a preconditioned system by solving:

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}\,, \tag{2.44}$$

where $\mathbf{M} \in \mathbb{R}^{n\times n}$ is the preconditioner which should be a positive definite matrix and an approximation of $\mathbf{A}$. In addition, it should also be inexpensive to invert for overall faster runtime.

Solving (2.44) leads to the Preconditioned Conjugate Gradient (PCG) algorithm, which is shown in Algorithm 1. We omit the details of preconditioned conjugate gradient. Interested readers are referred to [She+94].

As we aim to solve (2.41) with PCG, our target preconditioned system (2.44) becomes:

$$\mathbf{M}^{-1}\widetilde{\mathbf{H}}_{pp}(-\Delta\mathbf{x}_p) = \mathbf{M}^{-1}\widetilde{\mathbf{b}}_p\,. \tag{2.45}$$

Algorithm 1 Preconditioned Conjugate Gradient algorithm. t is a stopping threshold and $i_{\max}$ is the maximum number of iterations. We use a stopping criteria from [NS90; Nas00] in this thesis for better termination, in which $(q_{i+1} - q_i)$ computes the change in cost of the quadratic function defined by $\mathbf{A}$ and $\mathbf{b}$.

procedure PRECONDITIONED CONJUGATE GRADIENT($\mathbf{A}, \mathbf{b}, \mathbf{x}_0, \mathbf{M}^{-1}, t, i_{\max}$)
  $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
  $\mathbf{d}_0 = \mathbf{M}^{-1}\mathbf{r}_0$
  $\mathbf{q}_0 = -\mathbf{x}_0^\top(\mathbf{b} + \mathbf{r}_0)$
  for $i \leftarrow 0$ to $i_{\max}$ **do**
    $\alpha_i = \frac{\mathbf{r}_i^\top \mathbf{M}^{-1}\mathbf{r}_i}{\mathbf{d}_i^\top \mathbf{A}\mathbf{d}_i}$
    $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i \mathbf{d}_i$
    $\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i \mathbf{A}\mathbf{d}_i$
    $\beta_{i+1} = \frac{\mathbf{r}_{i+1}^\top \mathbf{M}^{-1}\mathbf{r}_{i+1}}{\mathbf{r}_i^\top \mathbf{M}^{-1}\mathbf{r}_i}$
    $\mathbf{d}_{i+1} = \mathbf{M}^{-1}\mathbf{r}_{i+1} + \beta_{i+1}\mathbf{d}_i$
    $q_{i+1} = -x_{i+1}^\top(b + r_{i+1})$
    if $\frac{i(q_{i+1} - q_i)}{q_{i+1}} < t$ **then**
      break
    end if
  end for
  return $\mathbf{x}_{i+1}$
end procedure

In this work, $\mathbf{M}$ is a block diagonal matrix with the diagonal blocks of $\widetilde{\mathbf{H}}_{pp}$, which is also known as the *Schur Jacobi* preconditioner.

Solving (2.41) only requires matrix-vector multiplications with RCS at each LM iteration. In Section 4.5, we introduce two options to perform matrix-vector multiplications with $\widetilde{\mathbf{H}}_{pp}$, which are explicitly and implicitly.

3 Methodologies

In this chapter, we explain the mathematical and implementation details of our solvers. We begin by introducing the concept of factor grouping and the grouping algorithm, and then discuss the differences between explicit and implicit RCS representations. Following that, we will discuss the mathematical details of square root formulation and its efficient implementation design. We present two new solvers: an implicit variant of the proposed square root solver and a square root solver collaborated with factor grouping scheme. Finally, we present a novel solver for solving RCS based on power series approximation of the inverse of the reduced camera matrix.

3.1 Factor Schur Complement

Factor grouping [Car+14] examines the structure of the BA problem using a factor graph. It demonstrates how to combine multiple factors into a single factor. Next, it compares the computational cost of explicit and implicit representations of multiplying with RCS, and then it demonstrates that a factor can be represented explicitly or implicitly. Following that, it investigates the criteria of selecting the most efficient factor presentation in terms of computing efficiency, allowing us to improve the multiplication efficiency in PCG.

In this section, we first describe the explicit SC and the implicit SC solver, which solve the reduced system using the explicit or implicit representation of RCS respectively. Then we examine the computational complexity of the both representations. Next, we briefly introduce the concept of factor graph in the context of bundle adjustment, then present Factor SC solver. After that, we describe *frequent pattern tree* (FP-tree), which is a data structure we used for grouping factors. Finally, we describe how Jacobian is efficiently stored and used for SC solvers.

3.1.1 Explicit and Implicit Representation

As the names suggests, the explicit representation explicitly constructs the RCS matrix $\widetilde{\mathbf{H}}_{pp}$ and it is stored in memory as a sparse matrix. In contrast, the implicit representation does not need to construct the whole matrix, but only the diagonal blocks for the PCG preconditioner and $\mathbf{H}_{ll}^{-1}$ for multiplication. We refer to solving the RCS

with the explicit representation as explicit SC solver and solving the RCS with implicit representation as the implicit SC solver.

Matrix-vector multiplication is slightly different for these two representations. For explicit SC, multiplication is performed with $\widetilde{\mathbf{H}}_{pp}$ directly. On the other hand, since RCS is made up of sub-blocks of $\mathbf{H}$ (2.37), implicit SC performs multiplication by sequentially accessing the sub-blocks of $\mathbf{H}$:

$$\mathbf{x} \leftarrow \underbrace{\mathbf{J}_p^\top \mathbf{J}_p}_{\mathbf{H}_{pp}} - \underbrace{\mathbf{J}_p^\top \mathbf{J}_l}_{\mathbf{H}_{pl}} \mathbf{H}_{ll}^{-1} \underbrace{\mathbf{J}_l^\top \mathbf{J}_p}_{\mathbf{H}_{lp}} \mathbf{x} \,. \tag{3.1}$$

For maximum efficient, the multiplication is reordered as:

$$\mathbf{x} \leftarrow \mathbf{J}_p^\top (\mathbf{J}_p \mathbf{x} - \mathbf{J}_l (\mathbf{H}_{ll}^{-1} (\mathbf{J}_l^\top (\mathbf{J}_p \mathbf{x})))) \,, \tag{3.2}$$

where we can reuse the result of $\mathbf{J}_p\mathbf{x}$ twice, and multiply with $\mathbf{J}_p^\top$ only once. In our implementation, it is evaluated efficiently as a parallel sum over the contributions of all landmarks.

3.1.2 Complexity Analysis of Explicit and Implicit Representation

In our experiments, multiplications with RCS during PCG typically consume the majority of runtime while solving a BA problem, it is critical to evaluate the computational complexity of both explicit and implicit representations.

Consider a BA problem with N_p cameras and N_l landmarks, we assume every landmark is observed by all cameras for simplicity. If two cameras observe at least one common landmark, then the blocks associated with the two cameras in RCS will be non-zero, therefore the RCS matrix is completely dense according to our assumption.

For explicit representation, a vector is multiplied directly with $\widetilde{\mathbf{H}}_{pp}$. Since the size of the matrix is $N_p \times N_p$, the multiplication complexity is then $O(N_p^2)$.

For implicit representation, a vector is multiplied with the sequence of matrices in (3.2). The complexity of multiplying with the block diagonal matrices $\mathbf{H}_{pp}$ and $\mathbf{H}_{ll}$ is $O(N_p)$ and $O(N_l)$ respectively. For $\mathbf{J}_p$ and $\mathbf{J}_l$, the complexity of multiplying with them is both $O(N_p N_l)$, which is the total number of observations. Consequently, the multiplication of complexity is $O(N_p N_l)$

Based on the time complexity, if an RCS includes more landmarks than cameras, i.e., $n_l > n_p$, explicit representation is preferred over implicit representation, and vice versa.

3.1.3 Factor Graph for Bundle adjustment

A factor graph is a bipartite graph represented by $\mathcal{G} = \{\mathcal{F}, \Theta\}$, where $\mathcal{F}$ denotes a set of factors and Θ denotes a set of variables. Each factor corresponds to a measurement,

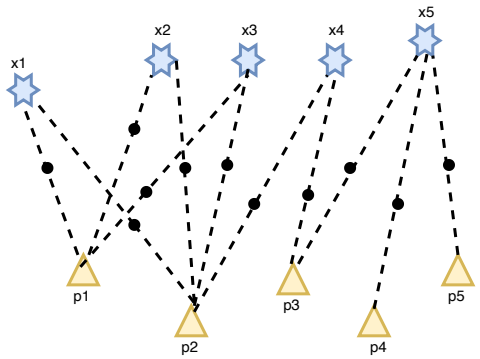


Figure 3.1: A factor graph on an example BA problem with 5 landmarks and 5 cameras [Car+14]. Landmarks are shown as blue stars, cameras are shown as yellow triangles.

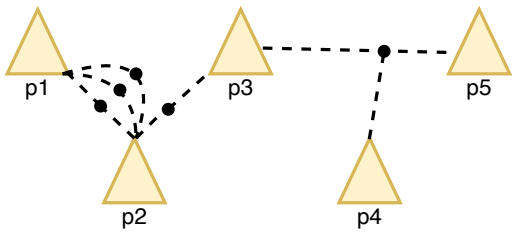


Figure 3.2: A factor graph of the problem in Figure 3.1 reduced by Schur complement trick. [Car+14]

and each variable to an optimizable parameter. A factor graph can be used to describe the BA objective function (2.22).

Figure 3.1 illustrates a factor graph of an BA example problem. In the context of BA, camera parameters and landmark variables are represented by variables Θ, and the observations are represented by factors $\mathcal{F}$. According to the reprojection error (2.21), each observation only involves a single camera and landmark. Consequently, a factor is always connected to a camera and a landmark variable.

By applying the Schur complement trick, landmarks are eliminated and leaving only the camera parameters in the reduced system (2.41). The factor graph depicting the reduced system is illustrated in Figure 3.2. In the graph, the Schur complement trick eliminates all the landmark variables and also the existing factors in the graph. Meanwhile, for each eliminated landmark, a new factor is created that connects to all camera variables observing that landmark.

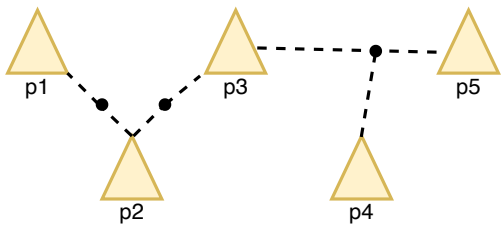


Figure 3.3: A grouped factor graph of the problem in Figure 3.2. [Car+14]

By representing a BA problem as a factor graph, we can notice that each factor independently contributes to the cost of the objective function (2.22). Subsequently, we can show the result of matrix-vector multiplication with $\widetilde{\mathbf{H}}_{pp}$ is the sum of all factor contributions:

$$\widetilde{\mathbf{H}}_{pp}\mathbf{x} = \sum_{i=1}^{n_l} \widetilde{\mathbf{H}}_{pp}^{i}\mathbf{x}\,, \tag{3.3}$$

where $\widetilde{\mathbf{H}}_{pp}^{i}$ is a zero-padded matrix which includes the contribution of factor i, it is zero everywhere except for the blocks corresponding to the cameras observing landmark i.

In Figure 3.2, there are three factors all connected to the same pair of cameras p_1, p_2. In fact, it is possible to group those factors together into a single one as illustrated in Figure 3.3, then this new factor now associates to three landmarks and two cameras.

According to (3.3), we can form an RCS representation for each factor. As a factor can be associated with multiple landmarks and cameras, we can select the best RCS representation of each factor based on time complexity introduced in Section 3.1.2.

For instance, as illustrated in Figure 3.3, the factor connected to p_1 and p_2 associates with 3 landmarks and 2 cameras. According to our analysis, since the number of landmarks is greater than cameras, explicit representation is preferred. On the other hand, implicit representation is preferred for the other factors.

The core idea of factor grouping [Car+14] is to group the landmarks into a number of factors. For the factors which associate with more landmarks than cameras, we explicitly construct a RCS matrix. For the remaining factors, they are considered as individual landmarks, and multiplication is performed via implicit representation. Then the result of multiplication with $\widetilde{\mathbf{H}}_{pp}$ is computed by accumulating the result of the factors and remaining landmarks as described in (3.3). As a result, by selecting the optimal representation between explicit and implicit, multiplication with RCS can be accelerated.

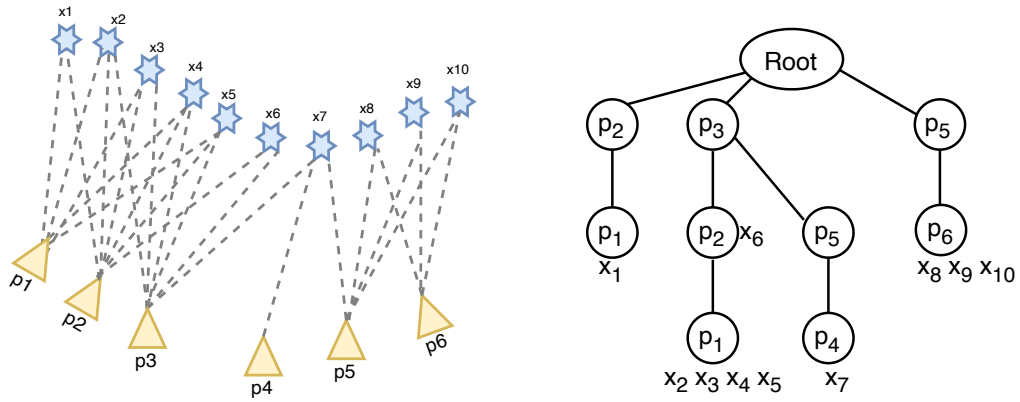


Figure 3.4: On the left is a BA example with 6 cameras and 10 landmarks. The corresponding grown FP-tree is shown on the right. [Car+14]

3.1.4 Frequent Pattern Tree

To group the factors, [Car+14] utilizes a data structure called *frequent pattern tree* (FP-tree) [HPY00], which can be used to mine *Frequent Itemsets* [Han+07]. In the context of Frequent Itemsets, each *transaction* can purchase a set of *items*. FP-tree can be used to identify groups of items that are frequently purchased together in the transactions database.

In our setting, a camera observes multiple landmarks, and we want to identify a list of factors by recognizing the set of landmarks that the same set of cameras frequently observes. Thereupon, cameras can be seen as items, and a landmark is a transaction involving multiple cameras. Based on this insight, FP-tree can be used group landmarks into factors.

As illustrated in Figure 3.4, FP-tree is a tree-like data structure. With the exception of the root node, which is an empty node, a node is associated with a camera and may contain zero or more landmarks. The path from root node to the node containing a landmark represents the list of observations of that landmark. For example in the figure, cameras p_4, p_5, p_3 are the observation of landmark x_7.

To grow the tree, the algorithm first sorts the list of observations of each landmark according to the *support* of cameras, which is the total occurrence of a camera. Then according to the camera support, a camera with larger support will be placed near the root node. In our example, since p_2 occurs more frequently than p_1, it is placed higher than p_1 on the leftmost branch.

We implement the factor grouping algorithm according to [Car+14]. It is mainly divided into two stages: identifying eligible factors, and trying to merge the remaining

landmarks into existing factors. In the first stage, we iterate all the leaf nodes. For each leaf node, we traverse up to the root and accumulate the landmarks along the path. By checking if the number of collected landmarks is greater than the number of cameras along the path, i.e. $n_l > n_p$, we can determine whether a new factor can be formed. At the second stage, we iterate each ineligible factor group from the first stage, i.e. the group with $n_l \leq n_p$. For each such group, we try to find a factor where the grouped landmarks are a superset of the ineligible group. If such a factor is found, the landmarks of the ineligible group are merged into the factor. Finally, we can obtain a list of factors that will employ explicit representation, and a list of remaining landmarks that will employ implicit representation.

We illustrate the result of the algorithm with Figure 3.4. In the first stage, we can form two factors: $\{x_2, x_3, x_4, x_5, x_6\}$ observed by $\{p_1, p_2, p_3\}$, and $\{x_8, x_9, x_{10}\}$ observed by $\{p_5, p_6\}$. At the second stage, we can notice that $\{p_1, p_2\}$ is a subset of $\{p_1, p_2, p_3\}$, therefore we can merge x_1 into the factor group $\{x_2, x_3, x_4, x_5, x_6\}$. At the end, we obtain two factor groups: $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ and $\{x_8, x_9, x_{10}\}$. During optimization, we use explicit representation for these two factor groups, and use implicit representation for the remaining landmark $\{x_7\}$.

Unlike the implementation in [Car+14], which runs in a single thread, our implementation of FP-tree is designed for parallelism. Our result in Section 4.6.1 demonstrates that our implementation is faster and is capable of efficiently operating on large BA problems. However, it yields a different outcome compared to the paper.

3.1.5 SC Landmark block

As represented in Figure 2.1, the Jacobian matrix $\mathbf{J}$ has a very sparse structure. Instead of storing the Jacobian as a dense matrix, it is stored compactly in *landmark block* to reduce memory usage. As illustrated in Figure 3.5, the Jacobian blocks in the same row which belongs to the same landmark, are stored in a dense matrix. For each landmark, we allocate a landmark block, and store the corresponding Jacobian blocks $\mathbf{J}_p, \mathbf{J}_l$ and residual in the landmark block.

Modern computers, which typically have multiple CPU cores and support SIMD instructions for efficient matrix operations. By utilizing landmark blocks, we can take advantages from these feature. Since landmark blocks are independent from each others during optimization, we can parallelize our operations over landmarks, e.g. construct $\widetilde{\mathbf{H}}_{pp}$, matrix-vector multiplication (3.2). Likewise, the operations performing on a landmark block can be supported by SIMD instructions as it is a dense matrix. Additionally, when the computations are parallelized over landmarks, a single landmark block stores all of the information necessary for solving via PCG and back-substitution closely in the memory, which improves the memory access pattern when the computations are

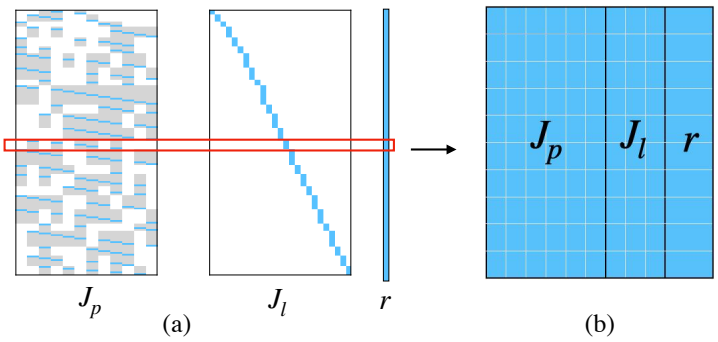


Figure 3.5: Jacobian and residual (a) related to the landmark are stores in a landmark block (b), which contains all the necessary information during optimization. Given a landmark with n observations, then the size of the landmark block is $2n \times (9+3+1)$. The camera blocks $\mathbf{J}_p$ which observes the current landmark are placed in a $2n \times 9$ block. The landmark blocks $\mathbf{J}_l$ and residual $\mathbf{r}$ are attached on the right. Adapted from [Dem+21].

parallelized over landmarks.

In the subsequent sections, we will refer this landmark block as SC landmark block as it is designed for storing Jacobian for SC solvers.

3.2 Square Root Bundle Adjustment

Square Root Bundle Adjustment ($\sqrt{BA}$) [Dem+21], is an alternative method to the Schur complement tick, which both aims to reduce the size of the normal equation (2.33). $\sqrt{BA}$ proposed a reformulation of (2.33), it relies on nullspace marginalization of the landmarks by QR decomposition. Comparing to Schur complement, the numeric stability is proved to be notable better than Schur complement in the experiments, thus allowing solving large-scale BA problems accurately even in single precision floating point format.

In the following sections, we introduce the mathematical and implementation details of $\sqrt{BA}$. First, we provide a high-level overview of QR decomposition and Givens rotations, which serve as the foundation for the new formulation. Following that, we detail the new square root formulation together with nullspace marginalization. Then we describe how the square root formulation is stored in memory during optimization. Following that, we describe how damping is accomplished with the new formulation. Finally, we present an implicit $\sqrt{BA}$ and a factor variant which we call $\sqrt{IBA}$ and

$\sqrt{FBA}$ in short.

3.2.1 QR Decomposition

QR decomposition is a matrix decomposition method, which decomposes a matrix into a product of an orthogonal matrix and upper triangular matrix. $\mathbf{A} \in \mathbb{R}^{m\times n}$, where $m \geq n$, is a matrix with rank n. It can be decomposed into an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{m\times m}$ and an upper triangular matrix $\mathbf{R} \in \mathbb{R}^{m\times n}$:

$$\mathbf{A} = \mathbf{QR} = \mathbf{Q}\begin{pmatrix}\mathbf{R}_1\\ \mathbf{0}\end{pmatrix} = \begin{pmatrix}\mathbf{Q}_1 & \mathbf{Q}_2\end{pmatrix}\begin{pmatrix}\mathbf{R}_1\\ \mathbf{0}\end{pmatrix} = \mathbf{Q}_1\mathbf{R}_1\,, \tag{3.4}$$

where $\mathbf{Q}$ is divided into $\mathbf{Q}_1 \in \mathbb{R}^{m\times n}$ and $\mathbf{Q}_2 \in \mathbb{R}^{m\times(m-n)}$, and $\mathbf{R}_1 \in \mathbb{R}^{n\times n}$ is an upper triangular matrix. (3.4) implies the columns of $\mathbf{Q}_2$ form the left nullspace of $\mathbf{A}$, i.e. $\mathbf{Q}_2^\top\mathbf{A} = \mathbf{0}$.

Besides, $\mathbf{Q}$ is an orthogonal matrix, which means:

$$\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}_m = \mathbf{Q}\mathbf{Q}^\top\,, \tag{3.5}$$

it can also be rewritten as:

$$\mathbf{Q}_1\mathbf{Q}_1^\top + \mathbf{Q}_2\mathbf{Q}_2^\top = \mathbf{I}_m. \tag{3.6}$$

3.2.2 Givens Rotations

A Givens rotation is a rotation in the plane spanned by two coordinates axes. We denote the orthogonal Givens rotation matrix as:

$$G_{ij}(\theta) = \begin{pmatrix}
1 & \cdots & 0 & \cdots & 0 & \cdots & 0\\
\vdots & \ddots & \vdots & & \vdots & & \vdots\\
0 & \cdots & \cos\theta & \cdots & \sin\theta & \cdots & 0\\
\vdots & & \vdots & \ddots & \vdots & & \vdots\\
0 & \cdots & -\sin\theta & \cdots & \cos\theta & \cdots & 0\\
\vdots & & \vdots & & \vdots & \ddots & \vdots\\
0 & \cdots & 0 & \cdots & 0 & \cdots & 1
\end{pmatrix}. \tag{3.7}$$

Givens rotation $G_{ij}(\theta)$ can be applied to a matrix $\mathbf{A}$ on the left, it rotates the ij-plane of $\mathbf{A}$, which only changes two rows in $\mathbf{A}$. Furthermore, by choosing the value of θ, we can

use a Given rotation to eliminate a entry $\mathbf{A}_{ij}$ to zero :

$$\begin{aligned} \cos\theta &= \frac{a_{jj}}{\sqrt{a_{jj}^2 + a_{ij}^2}}, \\ \sin\theta &= \frac{a_{ij}}{\sqrt{a_{jj}^2 + a_{ij}^2}}. \end{aligned} \tag{3.8}$$

Thereupon, the QR decomposition of $\mathbf{A}$ can be obtained by applying Given rotation matrices sequentially to $\mathbf{A}$, such that all entries below the diagonal are zero.

3.2.3 Nullspace Marginalization

To construct the new square root formulation, we factorize the landmark Jacobian $\mathbf{J}_l$ and project (2.32) onto the nullspace of $\mathbf{J}_l$. Similar to the previous sections, for simplicity we will omit the damping terms in the derivation.

We first construct the QR decomposition of $\mathbf{J}_l = \mathbf{QR}$. As $\mathbf{Q}$ is orthonormal, we can multiply the term in (2.32) with $\mathbf{Q}$ without changing the cost:

$$\begin{aligned} &\|\mathbf{r} + \begin{pmatrix}\mathbf{J}_p & \mathbf{J}_l\end{pmatrix} \begin{pmatrix}\Delta\mathbf{x}_p \\ \Delta\mathbf{x}_l\end{pmatrix} \|^2 \\ =&\|\mathbf{Q}^\top\mathbf{r} + \begin{pmatrix}\mathbf{Q}^\top\mathbf{J}_p & \mathbf{Q}^\top\mathbf{J}_l\end{pmatrix} \begin{pmatrix}\Delta\mathbf{x}_p \\ \Delta\mathbf{x}_l\end{pmatrix} \|^2 \\ =&\|\mathbf{Q}_1^\top\mathbf{r} + \mathbf{Q}_1^\top\mathbf{J}_p\Delta\mathbf{x}_p + \mathbf{R}_1\Delta\mathbf{x}_l\|^2 + \|\mathbf{Q}_2^\top\mathbf{r} + \mathbf{Q}_2^\top\mathbf{J}_p\Delta\mathbf{x}_p\|^2. \end{aligned} \tag{3.9}$$

Then we can solve $\Delta\mathbf{x}_p$ by optimizing the second term in (3.9):

$$\min_{\Delta\mathbf{x}_p} \frac{1}{2}\|\mathbf{Q}_2^\top\mathbf{r} + \mathbf{Q}_2^\top\mathbf{J}_p\Delta\mathbf{x}_p\|^2. \tag{3.10}$$

By constructing the normal equations of (3.10), we obtain:

$$\mathbf{J}_p^\top\mathbf{Q}_2\mathbf{Q}_2^\top\mathbf{J}_p(-\Delta\mathbf{x}_p) = \mathbf{J}_p^\top\mathbf{Q}_2\mathbf{Q}_2^\top\mathbf{r}, \tag{3.11}$$

which can be solved by PCG to obtain an optimal $\Delta\mathbf{x}_p^*$. Given $\Delta\mathbf{x}_p^*$, we can solve $\Delta\mathbf{x}_l$ using the first term in (3.9) by back substitution:

$$\Delta\mathbf{x}_l^* = -\mathbf{R}_1^{-1}(\mathbf{Q}_1^\top\mathbf{r} + \mathbf{Q}_1^\top\mathbf{J}_p\Delta\mathbf{x}_p^*). \tag{3.12}$$

Similar to Schur complement, this formulation substantially reduces the size of (2.32) by eliminating the landmarks. Additionally, (3.11) is proven to be algebraically equivalence to Schur complement trick (2.41):

$$\begin{aligned} \widetilde{\mathbf{H}}_{pp} &= \mathbf{J}_p^\top\mathbf{Q}_2\mathbf{Q}_2^\top\mathbf{J}_p, \\ \widetilde{\mathbf{b}}_p &= \mathbf{J}_p^\top\mathbf{Q}_2\mathbf{Q}_2^\top\mathbf{r}, \end{aligned} \tag{3.13}$$

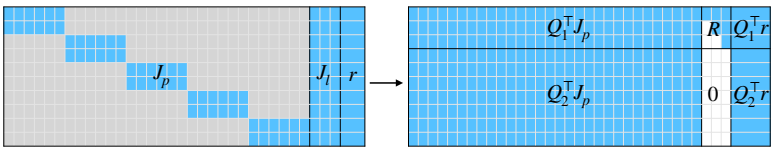


Figure 3.6: A $\sqrt{BA}$ landmark block (a) stores the all necessary information during optimization [Dem+21]. Landmarks can be marginalized by QR decomposition (b). Given a landmark with n observation, the size of $\sqrt{BA}$ landmark block is $2n \times (9n + 3 + 1)$.

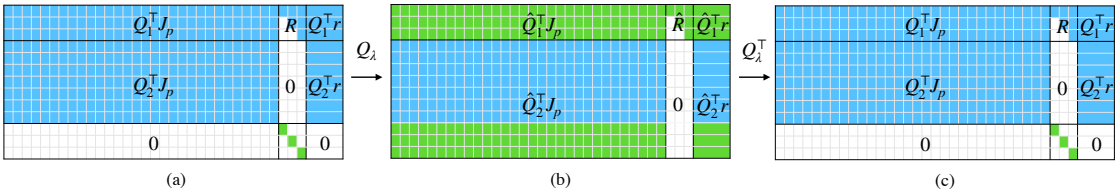


Figure 3.7: Damping landmark with a $\sqrt{BA}$ landmark block after marginalization [Dem+21].

in which $\mathbf{Q}_2^\top \mathbf{J}_p$ is the square root form of RCS. Similarly, the back substitution for optimal $\Delta \mathbf{x}_p^*$ (3.12) is also algebraically equivalence to (2.42).

3.2.4 Damping with Levenberg-Marquardt

In Schur complement, damping λ is added to the RCS as:

$$\widetilde{\mathbf{H}}_{pp}^{\lambda} = (\mathbf{H}_{pp} + \lambda \mathbf{D}_p^\top \mathbf{D}_p) - \mathbf{H}_{pl}(\mathbf{H}_{ll} + \lambda \mathbf{D}_l^\top \mathbf{D}_l)^{-1} \mathbf{H}_{lp} \tag{3.14}$$

$$\widetilde{\mathbf{b}}_p^{\lambda} = \mathbf{b}_p - \mathbf{H}_{pl}(\mathbf{H}_{ll} + \lambda \mathbf{D}_l^\top \mathbf{D}_l)^{-1} \mathbf{b}_l \,, \tag{3.15}$$

where $\mathbf{D}_p^\top \mathbf{D}_p$ is the diagonal of $\mathbf{H}_{pp}$ and $\mathbf{D}_l^\top \mathbf{D}_l$ is the diagonal of $\mathbf{H}_{ll}$.

A possible solution for $\sqrt{BA}$ is attaching a diagonal matrix $\lambda \mathbf{D}_l$ under $\mathbf{J}_l$, and then compute the QR decomposition on the augmented $\mathbf{J}_l$. However, this will significantly increase runtime, as QR decomposition has a time complexity of $O(n^3)$.

Instead, we again work with the individual landmark blocks. We propose $\sqrt{BA}$ landmark block as shown in Figure 3.6, which is based on SC landmark block in Section 3.1.5. The only difference from SC landmark block is that the camera Jacobian blocks are stored diagonally in the $\sqrt{BA}$ landmark block. To perform marginalization with $\sqrt{BA}$ landmark block, we need to apply a sequence a Givens rotations to the landmark

block in-place. We attach a 3×3 sub-block to the bottom of $\mathbf{Q}^\top \mathbf{J}_p$, then a landmark block can be eliminated by six Givens rotations as illustrated in Figure 3.7. We denote the damped $\mathbf{Q}$ as:

$$\hat{\mathbf{Q}} = (\hat{\mathbf{Q}}_1 \quad \hat{\mathbf{Q}}_2) = \begin{pmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \lambda \mathbf{I}_3 \end{pmatrix} \mathbf{Q}_\lambda \,, \tag{3.16}$$

where $\mathbf{Q}_\lambda$ is the product of the six Givens rotations.

As shown in Figure 3.6, the six Givens rotations eliminate the bottom part of $\mathbf{J}_l$ to zeros. After that, we can obtain then an upper triangle matrix $\mathbf{R} \in \mathbb{R}^{3\times3}$. Simultaneously, as we apply Given rotations to the whole landmark block, we can also obtain $\hat{\mathbf{Q}}^\top \mathbf{J}_p$ and $\hat{\mathbf{Q}}^\top \mathbf{r}$ at the end.

If LM algorithm rejects the update and increase the value of λ, we do not have to discard the landmark blocks and recompute them with a new λ. Instead, we can remove the old damping by applying the inverse of the six Givens rotations, then apply a new damping with again six Givens rotations. Consequently, damping the landmark is computationally cheap by applying on the individual landmark blocks.

For pose damping, we can just sum the multiplication with $\lambda \mathbf{D}_p^\top \mathbf{D}_p$ from (3.16) during PCG:

$$\mathbf{x} \leftarrow \mathbf{J}_p^\top \hat{\mathbf{Q}}_2 \hat{\mathbf{Q}}_2^\top \mathbf{J}_p \mathbf{x} + \lambda \mathbf{D}_p^\top \mathbf{D}_p \mathbf{x} \,. \tag{3.17}$$

3.2.5 Implicit Square Root Bundle Adjustment

In this work, we extend $\sqrt{BA}$ with an implicit variant named as $\sqrt{IBA}$, which consumes significant less memory than $\sqrt{BA}$. As illustrated in Figure 3.6, because the camera Jacobian blocks are stored diagonally in the landmark block, $\sqrt{BA}$ requires more memory that scales quadratically to the number of observations of a landmark. As a result, if a bundle project is dense, i.e. landmarks are observed by a large number of cameras, $\sqrt{BA}$ will consume significantly more memory.

For each LM iteration, $\sqrt{BA}$ first linearizes the problem, then performs marginalization. After that, for each PCG iteration, $\hat{\mathbf{Q}}_2^\top \mathbf{J}_p$ is recalled from the memory to perform multiplications. In contrast, $\sqrt{IBA}$ does not store $\hat{\mathbf{Q}}_2^\top \mathbf{J}_p$ but compute it on-the-fly per PCG iteration with the SC landmark block introduced in Section 3.1.5.

For each LM iteration, $\sqrt{IBA}$ linearizes the problem and store the Jacobian in SC landmark blocks. Then, for each PCG iteration, we allocate a new $\sqrt{BA}$ landmark block and copy the data from the SC landmark block to it. Next, we dampen and marginalize the new block, and use $\hat{\mathbf{Q}}_2^\top \mathbf{J}_p$ for multiplication in PCG.

To avoid reallocating memory for $\sqrt{BA}$ landmark blocks per PCG iteration, $\sqrt{IBA}$ allocates a dedicated chunk of memory for marginalizing. At the beginning, we allocate a volume of memory that can fit the largest $\sqrt{BA}$ landmark block. Then for each PCG

iteration, the Jacobian is copied from the SC landmark to the dedicated memory, then marginalization is performed there. Since the algorithm is run parallelized, we allocate a chunk of memory for each thread. As a result, We avoid allocating a large chunk of memory for $\sqrt{BA}$ landmark blocks.

Note that we use Householder reflection method instead of Givens rotation for QR decomposition. In our experiments, Householder is slightly faster than Given rotations. This could be because Householder needs less computations. To decompose $\mathbf{J}_l$ per landmark block, Householder requires only three transformations, whereas Given rotations require three transformations per row in a landmark block.

3.2.6 Factor Square Root Bundle Adjustment

Furthermore, we also develop a variant called $\sqrt{FBA}$, it exploits the factor group concept with the square root formulation. As introduced in Section 3.1, factor grouping allows grouping landmarks into factors, then we can represent the factors with explicit RCS representation during optimization, and use implicit representation for the remaining landmarks. It allows the landmark select the best representation in terms of computation complexity.

For $\sqrt{FBA}$, we compute the RCS matrix according to square root formulation (3.13). When explicit RCS matrix is used, the landmark blocks are only used when constructing RCS matrix and back substitution. Hence same as $\sqrt{IBA}$, we store Jacobian in SC landmark blocks to be memory efficient. On the other hand, $\sqrt{IBA}$ is tested slightly faster than $\sqrt{BA}$, therefore we also employ $\sqrt{IBA}$ scheme for non factor landmarks.

In summary, same as $\sqrt{IBA}$, $\sqrt{FBA}$ stores Jacobian in SC landmark blocks. For each factors, we construct a RCS matrix for PCG multiplication using square root formulation. For non-factor landmarks, we again use $\sqrt{IBA}$ for multiplications in PCG.

3.3 Power Bundle Adjustment

In the previous sections, we describe the various iterative solvers, which all rely on performing matrix-vectormultiplication with the RCS in PCG. Alternatively, inspire from [ZXS21], we develop a solver that does not solve the reduce system using iterative methods like PCG, but approximate the inverse of $\widetilde{\mathbf{H}}_{pp}$ using power series expansion. We name the new solver as power BA [Web+22].

In the following, we first introduce the mathematical formulation of power BA, and then the implementation details.

3.3.1 Mathematical Formulation

Let M be a $n \times n$ square matrix. $(\mathbf{I} - \mathbf{M})^{-1}$ can be expressed as a power matrix series:

$$(\mathbf{I} - \mathbf{M})^{-1} = \sum_{i=0}^{\infty} \mathbf{M}^i , \tag{3.18}$$

if all eigenvalues of $\mathbf{M}$ are strictly between -1 and 1.

We can reformulate the RCS (2.40) as:

$$\begin{aligned} \widetilde{\mathbf{H}}_{pp} &= \mathbf{H}_{pp} - \mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp} \\ &= \mathbf{H}_{pp}(\mathbf{I} - \mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp}) , \end{aligned} \tag{3.19}$$

then its inverse can be written as:

$$\widetilde{\mathbf{H}}_{pp}^{-1} = (\mathbf{I} - \mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp})^{-1}\mathbf{H}_{pp}^{-1} \tag{3.20}$$

We can prove that the largest eigenvalue of $\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp}$ is between -1 and 1, which satisfies the condition to express as a power matrix series. We will skip the proof here. We then can approximate the inverse Schur complement at order m:

$$\tilde{\mathbf{H}}_{pp}^{-1} \approx \sum_{i=0}^{m} (\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp})^i \mathbf{H}_{pp}^{-1} \tag{3.21}$$

Accordingly, the optimal $\Delta\mathbf{x}_p^*$ can be approximated as:

$$\Delta\mathbf{x}_p^* \approx \sum_{i=0}^{m} (\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp})^i \mathbf{H}_{pp}^{-1}(-\widetilde{\mathbf{b}}_p) , \tag{3.22}$$

and then $\Delta\mathbf{x}_l^*$ can be solved by back substitution (2.42).

3.3.2 Implementation Details

Efficient computation

Although the equation (3.22) appears to involve with matrix-matrix multiplications $(\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp})^i$, we do not have to construct the matrix product explicitly. Similarly to the implicit SC solver, we can solve $\Delta\mathbf{x}_p^*$ efficiently by performing matrix-vector multiplications sequentially with the sub-blocks of $\mathbf{H}$.

$$\begin{aligned} \tilde{\mathbf{H}}_{pp}^{-1}\widetilde{\mathbf{b}}_p = \; & \mathbf{H}_{pp}^{-1}\widetilde{\mathbf{b}}_p + \\ & (\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp})\mathbf{H}_{pp}^{-1}\widetilde{\mathbf{b}}_p + \\ & (\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp})^2\mathbf{H}_{pp}^{-1}\widetilde{\mathbf{b}}_p + \cdots \end{aligned} \tag{3.23}$$

As demonstrated in (3.23), starting from the second term, each term can be computed based on the result of the previous term. For instance, the second item can be obtained by multiplying the first term with $\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp}$, which holds true for subsequent terms as well. As a result, the approximation (3.22) is accumulated from the result of a sequence of matrix-vector multiplication with the Jacobians.

Besides the efficient computation, different from implicit SC solver which can be heavily influenced by preconditioner [Aga+10], power BA produces stable results without a preconditioner as shown in Section 4.9. Since constructing the preconditioner can consume considerable amount of runtime, this can be very advantageous to the solvers relying on PCG.

Termination criteria

The order m is critical for both runtime and accuracy. If m is too low, the solution will be inaccurate; if m is too high, additional runtime will be incurred.

It can be shown that the following sequence converges to zero:

$$\{(\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp})^i\mathbf{H}_{pp}^{-1}\widetilde{\mathbf{b}}_p\}_{i=0}^{\infty}\,. \tag{3.24}$$

We can utilize a straightforward technique for dynamically selecting m. When we realize that increasing order has a negligible contribution to the solution, we can stop increasing m. The following termination criteria can be adopted:

$$\frac{\|(\mathbf{H}_{pp}^{-1}\mathbf{H}_{pl}\mathbf{H}_{ll}^{-1}\mathbf{H}_{lp})^i\mathbf{H}_{pp}^{-1}\widetilde{\mathbf{b}}_p\|}{\|\mathbf{H}_{pp}^{-1}\widetilde{\mathbf{b}}_p\|} < \epsilon\,, \tag{3.25}$$

where ϵ is a given parameter.

4 Evaluation

In this work, we develop a total of seven solvers in order to examine their performance. There are three main types of solvers: SC solvers (explicit, implicit, and factor SC) which use Schur complement trick; square root solvers ($\sqrt{IBA}$, $\sqrt{BA}$, and $\sqrt{FBA}$) which employ the square root formulation; power BA which approximates the inverse of $\widetilde{\mathbf{H}}_{pp}$ as a power series.

We evaluate the performance of our solvers comprehensively on three selected datasets. They cover a variety of BA problems, from small- to large-scale, and from sparse to dense problems.

Our research is mostly centered on experiments executed in double precision. In our experiments, we notice that repeatedly solving a problem in single precision yields slightly different results each time, whereas the solvers are more numerically stable in double precision, resulting in more reproducible results.

4.1 Datasets

We evaluate the performance of our solvers comprehensively on three distinct datasets, they are from Bundle Adjustment in the Large (BAL) [Aga+10], 1DSfM [WS14], and Multicore Bundle Adjustment (MCBA) [Wu+11] respectively. They are all constructed using a collection of internet images.

Despite each dataset was stored in a specific format, they all adopted the same camera model described Section 2.1.2. The BA problems in all dataset always contain the following information:

- 3D landmark coordinates $\{\mathbf{X}_i\}_{i=1}^{n_l}$,
- Camera parameters $\{\mathbf{R}_j, \mathbf{t}_j, f_j, \mathbf{k}_j\}_{j=1}^{n_p}$,
- The pixel location $\mathbf{u}_{ij}$ of landmark i observed by camera j,
- The list of observations of each landmark.

4.1.1 Bundle Adjustment in the Large

Bundle Adjustment in the Large (BAL) dataset has a total 97 BA problems. They are divided into different groups: Ladybug, Trafalgar, Dubrovnik, Venice, and Final. We name a problem according to its group and number of cameras, e.g. final4585. The problems in each group have slightly distinct characteristics. Ladybug problems are captured by a moving Ladybug camera, thus they are sparser than other groups. Trafalgar, Dubrovnik, and Venice are reconstructed using internet image collections, each looking at a particular scene.

The problems in the aforementioned groups are reconstructed by Bundler [SSS06], it reconstructs by incrementally adding images in each iteration. Trafalgar, Dubrovnik, and Venice serve as the skeletons of the Final problems, which are built by adding more images to the skeleton. As a result, the final problems often have a significantly larger camera count and are denser than other groups.

BAL dataset contains small to large-scale problems, Table 4.1 shows the details of some of problems. For the full list of problems, interested readers are referred to the supplementary of [Dem+21]. BAL covers a wider range of problems than the other two datasets thus more representative, therefore we mainly perform our analysis on this dataset.

However, the problems in this dataset are relatively easy to solve. In our experiments, all our solvers can reduce the cost with fewer iterations in comparison to the other two datasets.

4.1.2 1DSfM

[WS14] proposes optimizing BA problems utilizing rotation and translation averaging. A dataset is created to evaluate the performance of the proposed algorithm, we refer it as 1DSfM dataset. Each problem in the dataset reconstructs a famous sight location from around the world. The dataset contains 15 problems, which are mostly small-scale problems. The details are shown in Table 4.2.

Since the dataset is stored as a different format, we utilize TheiaSfM [Swe] to convert the problems to Bundler format via the global SfM pipeline, which can then be imported into to our program.

4.1.3 Multicore Bundle Adjustment

[Wu+11] examines how parallelism on multiple CPU cores or GPU improves the performance of implicit Schur complement. The Multicore Bundle Adjustment (MCBA) dataset is built to evaluate the proposed solver.

	#cam	#lm	#obs	sparsity	#obs / cam	#obs / lm		
	(n_p)	(n_l)	(N_r)	(0 in $\widetilde{\mathbf{H}}_{pp}$)	mean	mean	std-dev	max
ladybug73	73	11,022	46,091	28%	631.4	4.2	3.7	40
ladybug539	539	65,208	277,238	71%	514.4	4.3	4.7	142
ladybug1723	1,723	156,410	678,421	92%	393.7	4.3	5.0	145
trafalgar126	126	40,037	148,117	62%	1,175.5	3.7	3.0	29
trafalgar170	170	49,267	185,604	67%	1,091.8	3.8	3.5	41
trafalgar257	257	65,131	225,698	76%	878.2	3.5	3.2	42
dubrovnik88	88	64,298	383,937	13%	4,362.9	6.0	6.0	65
dubrovnik173	173	111,908	633,894	37%	3,664.1	5.7	6.7	84
dubrovnik356	356	226,729	1,254,598	46%	3,524.2	5.5	6.4	122
venice89	89	110,973	562,976	7%	6,325.6	5.1	5.9	62
venice744	744	542,742	3,054,949	58%	4,106.1	5.6	8.6	205
venice1778	1,778	993,101	4,997,555	84%	2,810.8	5.0	7.1	232
final93	93	61,203	287,451	0%	3,090.9	4.7	5.8	80
final394	394	100,368	534,408	6%	1,356.4	5.3	10.6	280
final871	871	527,480	2,785,016	60%	3,197.5	5.3	9.8	245
final961	961	187,103	1,692,975	1%	1,761.7	9.0	29.3	839
final1936	1,936	649,672	5,213,731	3%	2,693.0	8.0	26.9	1293
final3068	3,068	310,846	1,653,045	79%	538.8	5.3	12.6	414
final4585	4,585	1,324,548	9,124,880	83%	1,990.2	6.9	12.6	535
final13682	13,682	4,455,575	28,973,703	86%	2,117.7	6.5	18.9	1748

Table 4.1: Size of the bundle adjustment problem for an exemplar subset of the BAL dataset.

	#cam	#lm	#obs	sparsity	#obs / cam	#obs / lm		
	(n_p)	(n_l)	(N_r)	(0 in $\widetilde{\mathbf{H}}_{pp}$)	mean	mean	std-dev	max
trafalgar	5,032	388,924	1,826,007	88%	362.9	4.7	11.1	327
alamo	571	151,084	891,299	22%	1,560.9	5.9	12.4	317
ellisisland	234	29,163	130,901	33%	559.4	4.5	5.6	121
gendarmenmarkt	706	93,667	364,019	81%	515.6	3.9	4.7	104
madridmetropolis	346	55,628	255,885	50%	739.6	4.6	6.0	143
montrealnotredame	459	158,002	860,110	40%	1,873.9	5.4	10.4	264
notredame	547	273,588	1,534,743	23%	2,805.7	5.6	11.1	308
nyclibrary	338	74,248	303,953	59%	899.3	4.1	5.3	93
piazzadelpopolo	335	37,605	195,008	43%	582.1	5.2	8.6	159
piccadilly	2,289	209,489	999,848	81%	436.8	4.8	8.9	283
romanforum	1,063	265,044	1,292,750	83%	1,216.1	4.9	7.4	151
toweroflondon	483	151,327	797,020	71%	1,650.1	5.3	6.2	90
unionsquare	796	46,057	230,793	86%	289.9	5.0	6.8	91
viennacathedral	836	265,553	1,333,279	70%	1,594.8	5.0	8.5	218
yorkminster	422	152,589	701,985	60%	1,663.5	4.6	6.0	126

Table 4.2: Size of the bundle adjustment problem for each instance in the 1DSfM dataset.

	#cam	#lm	#obs	sparsity	#obs / cam	#obs / lm		
	(n_p)	(n_l)	(N_r)	(0 in $\widetilde{\mathbf{H}}_{pp}$)	mean	mean	std-dev	max
01biltmore	26	5,215	20,901	0%	803.9	4.0	3.2	23
02notredame	71	16,793	78,734	13%	1,108.9	4.7	4.8	45
03notredame	1,290	35,324	614,978	18%	476.7	17.4	53.3	743
04liberty	9,022	23,910	1,681,838	24%	186.4	70.3	221.9	4213
05liberty	9,025	34,209	2,024,119	18%	224.3	59.2	200.4	4751
06bandenburger	4,652	31,194	1,889,850	2%	406.2	60.6	202.3	3232
07berlin	5,333	34,961	2,201,273	2%	412.8	63.0	227.4	3612
08rome	8,140	21,536	2,387,050	2%	293.2	110.8	377.1	6372
09rome	6,983	49,983	3,078,434	54%	440.8	61.6	141.1	1752
10sanmarco	10,338	65,326	3,416,834	67%	330.5	52.3	149.6	3126

Table 4.3: Size of the bundle adjustment problem for each instance in the MCBA dataset.

MCBA has a total of 10 problems, which are mostly large-scale problems as demonstrated in Table 4.3. Given the larger scale of this dataset, the efficiency of a solver can be highlighted by evaluating on it. Additionally, the average number of landmark observations is substantially higher than in the other two datasets. If a solver involves quadratic-complexity operations, it can be extremely slow on this dataset.

4.2 Experiment Setup

We adopt the configuration specified in [Dem+21], all our experiments use the same configuration if not specific stated. We apply preliminary preprocessing prior to optimizing a problem. A small amount of noise is added to the state for perturbation, and then invalid observations and under-constraint landmarks are removed.

For each problem, our solvers run 20 LM iterations, all solvers except power BA run a maximum of 500 PCG iterations per LM iteration. Power BA always solves a problem with order of 10. The damping parameter λ has an initial value of 10^{-4}, and it is adjusted according to the ratio of actual and estimated cost decrease per LM iteration. PCG terminates if the stopping criteria described in Algorithm 1 are satisfied. All solvers except power BA solve problem via PCG, we use the diagonal blocks of $\widetilde{\mathbf{H}}_{pp}$ as preconditioner [Aga+10]. On the other hand, power BA does not require any preconditioner because it does not use PCG to solve a problem.

All experiments are run on an Ubuntu 18.04 desktop. It equips with an Intel Xeon W-2133 CPU with 12 virtual cores running at 3.60GHz, along with 64GB of RAM. All major operations in our implementation are parallelized by using Intel TBB [Phe08], and for the dense linear algebra operations we relied on Eigen [G+10] compiled with the support of SIMD vectorization.

4.3 Performance Profiles

Given that the selected datasets are constructed using images retrieved from the internet, it is impossible to determine the ground truth of a reconstruction. Instead, we use the BA optimization cost (2.22) as an indicator of optimization accuracy. Given that the objective function should reflect the quality of our reconstruction, a lower cost often implies a better optimization result.

Performance is typically a trade-off between accuracy and speed for optimization solvers. Furthermore, given problems with different characteristic, a solver may perform better on some problems and worse on others. To provide an overview of the performance of multiple solvers, we use performance profiles [DM02] to visualize the accuracy and runtime of different solvers on multiple problems.

We provide a quick overview of performance profiles in the following. Let $\mathcal{P}$ be a set of BA problems, $\mathcal{S}$ be the set of solvers we evaluate on $\mathcal{P}$. Given a problem $p \in \mathcal{P}$ and a solver $s \in \mathcal{S}$, we define $f(p, s, t)$ as the function of minimum cost achieved by the solver at time t. We denote the initial cost of a problem p as $f_0(p)$. The minimum cost accomplished across all solvers in any time is defined as:

$$f^*(p) := \min_{s,t} f(p, s, t)\,. \tag{4.1}$$

Then, we define the cost threshold corresponding to a given accuracy tolerance $\tau \in (0, 1)$ as:

$$f_\tau(p) := f^*(p) + \tau(f_0(p) - f^*(p))\,. \tag{4.2}$$

For instance, $f_{0.1}(p)$ defines the cost value that has been reduced by 90% from the initial cost to the lowest possible cost.

The runtime of a solver s reaching cost of an accuracy tolerance $f_\tau(p)$ is denoted as:

$$t_\tau(p, s) := \min\{t \mid f(p, s, t) \leq f_\tau(p)\} \cup \{\infty\}\,, \tag{4.3}$$

where infinity is included since a solver may never reach the accuracy tolerance.

Finally, the performance profile of a solver s is defined as:

$$\rho_\tau(s, \alpha) := \frac{|\{p \mid t_\tau(p, s) \leq \alpha \min_s t_\tau(p, s)\}|}{|\mathcal{P}|} \cdot 100\,, \tag{4.4}$$

which essentially means the percentage of problems that solver s has reduced to the tolerance τ, at time $\alpha \min_s t_\tau(p, s)$ for individual problems.

Figure 4.1 shows such performance profiles. Consider the tolerance plot with $tau = 0.1$ on the left, power BA (power-64) is the fastest solver for reducing the cost with 90% of the problems by looking at $\alpha = 1$. On the other hand, $\sqrt{IBA}$ is hardly the fastest

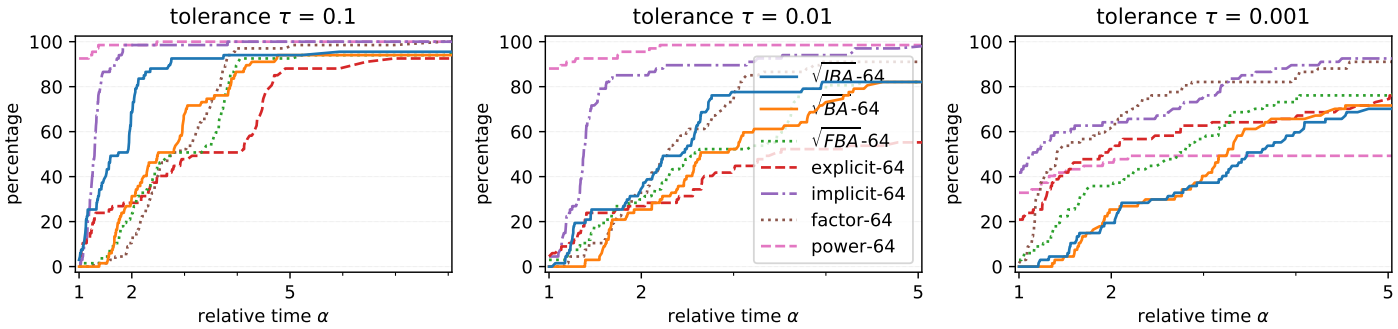


Figure 4.1: Performance profiles evaluated on BAL problems in double precision. X-axis is the percentage of problems that has reached the accuracy tolerance, y-axis is the relative time α.

solver, it only achieves the tolerance for 50% of the problems using twice the runtime ($\alpha = 2$) compared with fastest solver on the individual problems.

If the curve of solver is further to the left (lower α), this indicates the solver has a faster run time in terms of cost reduction. If the curve is closer to the top (higher percentage), it means the solver achieves the tolerance on more problems.

Note that we only illustrate the relative time α up to a certain number in the performance profiles. Thus, if the curve of a solve does not reach 100% in the plots, it either indicates the solver reaches the accuracy threshold beyond the maximum relative time of the plot, or it never reaches the threshold.

4.4 Performance Overview

4.4.1 Experiments on the BAL dataset

Double precision Figure 4.1 shows the performance profiles evaluated on BAL problems in double precision. In the following, we summarize the performance of each solver at each tolerance level.

At high tolerance $\tau = 0.1$, we can see power BA is overall the fastest solver, where the curve is close to 100% at $\alpha = 1$. Compared to power BA, implicit SC requires around 20% to 30% more time to accomplish the same tolerance. Following that, the speed $\sqrt{IBA}$ closely matches implicit SC on 40% of the problems, but it is around 60% slower than implicit SC on the remaining problems on average. Furthermore, the curve of $\sqrt{IBA}$ is lower than the ones of power BA and implicit SC, which indicates it cannot achieve the same level of precision on several problems as power BA and implicit SC

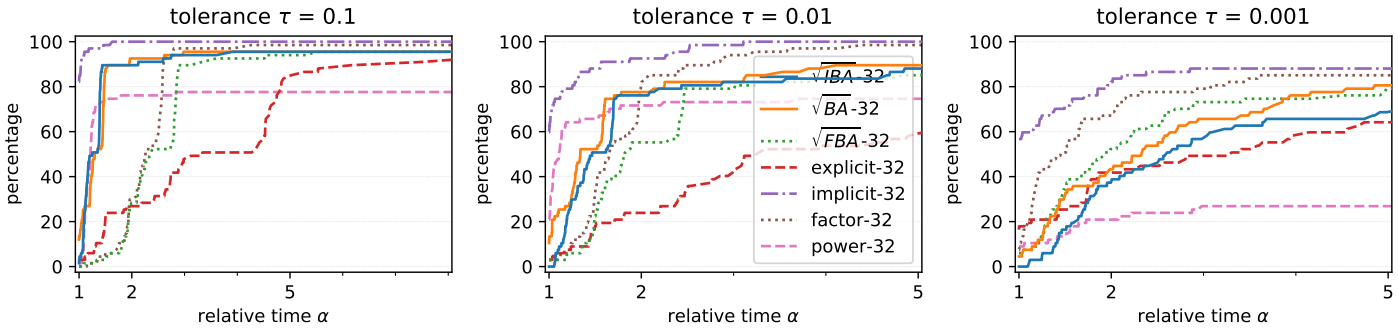


Figure 4.2: Performance profiles evaluated on BAL problems in single precision.

even after a long time.

$\sqrt{BA}$, $\sqrt{FBA}$, and explicit SC solver reach a similar accuracy as $\sqrt{IBA}$ but at a slower rate. While factor SC solver has a similar runtime to these three solvers, it accomplishes a higher accuracy as power BA and implicit SC after certain time. At tolerance $\tau = 0.01$, the situation is similar to the previous plot, except that $\sqrt{IBA}$ is notably slower and factor SC is comparably faster.

At low tolerance $\tau = 0.001$, the landscape is different from the previous tolerances. High tolerance emphasizes more on the speed of a solver, and low tolerance emphasizes more on accuracy. At $\alpha = 1$, implicit SC is the best performer, closely followed by power BA and explicit SC.

Compared to higher tolerance, power BA only achieves the accuracy on only 50% of the problems. This either suggests that power BA has a lower accuracy, or require a higher order to solve a problem precisely, given the order was fixed at 10, which could be inadequate for low tolerance.

The performance of factor SC, $\sqrt{FBA}$, and explicit SC solvers perform noticeably better at lower tolerance. In Section 4.5, our investigation shows explicit representation has a cheaper multiplication, and PCG requires more iterations at low tolerance. As a result, solvers using explicit representation gain more benefits at lower tolerance. Similarly, in Section 4.7, we demonstrate that multiplication of $\sqrt{IBA}$ is more expensive than $\sqrt{BA}$, therefore $\sqrt{IBA}$ is also slower at low tolerance.

In our results, power BA and implicit SC are overall the fastest solvers. In Section 4.8, we discuss the performance impact of a better memory access pattern, which could be the reason why they have an excellent performance. In Section 4.9, we examine the performance of power BA and compare it to implicit SC solver.

Single precision The performance of solvers in single precision is illustrated in Figure 4.2. In comparison to double precision, solving in single precision is numerically less accurate and contains more randomness in the result. As a result, solving a problem in single precision challenges the numerical properties of a solver.

Compared to result in double precision, power BA is no longer the top performer, the accuracy is significantly degraded across all tolerances. At tolerance $\tau = 0.1$, it is always slower than implicit SC indicated at $\alpha = 1$ and only solves 80% of the problems to the tolerance, which is the lowest among all solvers. This suggests Schur complement and square root formulation have better numerical properties than power BA, thus running power BA with single precision is less preferable. Moreover, power BA solves only 20% of problems to the tolerance $\tau = 0.001$, which suggests the fixed order of 10 may not have sufficient accuracy to solve a problem.

In contrast, $\sqrt{BA}$ and $\sqrt{IBA}$ perform considerably better compared to result in double precision. As demonstrated in [Dem+21], the square root formulation has a more favorable numerical properties, and hence suffers less from the drawbacks of single precision. For $\sqrt{FBA}$, since most of the landmarks are explicitly represented as a RCS matrix, therefore it receives similar accuracy limitations as the SC solvers.

Performance on problems with varied characteristic

The performance profile summarizes the performance of our solvers, but does not provide insight into their performance on a particular type of problem. In Figure 4.3, we examine the performance on problems with varied sparsity and problem scale.

All plots draw the same set of problems, each of which is either denoted by a colored dot, red diamond, or a pink cross. If a solver is the fastest to reduce the cost of a problem to a tolerance, then the problem in the corresponding plot is marked with a red diamond. If a solver is unable to solve a problem to the specified tolerance due to its accuracy limitation, or it runs out of memory during optimization, then the problem is indicated by a pink cross. Otherwise, we indicate the convergence rate of a solver solving a problem by a colored dot, which encodes the relative time α up to 3.0. The same problem is compared with other solvers at the same tolerance. Illustrated with the color scale on the right, a problem is shown in purple if the speed is the fastest, and in yellow if it is slower.

In the figure, as with the performance problem, power BA converges very quickly at higher tolerances $\tau = \{0.1, 0.01\}$, but it is not able to solve most problems accuracy at low tolerance $\tau = 0.001$. Implicit SC performs fairly similar to power BA, it is very close to power BA across all tolerance levels. At tolerance $\tau = 0.001$, it becomes fastest on larger problems as power BA is less accurate at low tolerance.

On the other hand, explicit SC performs overall better on small to medium-scale

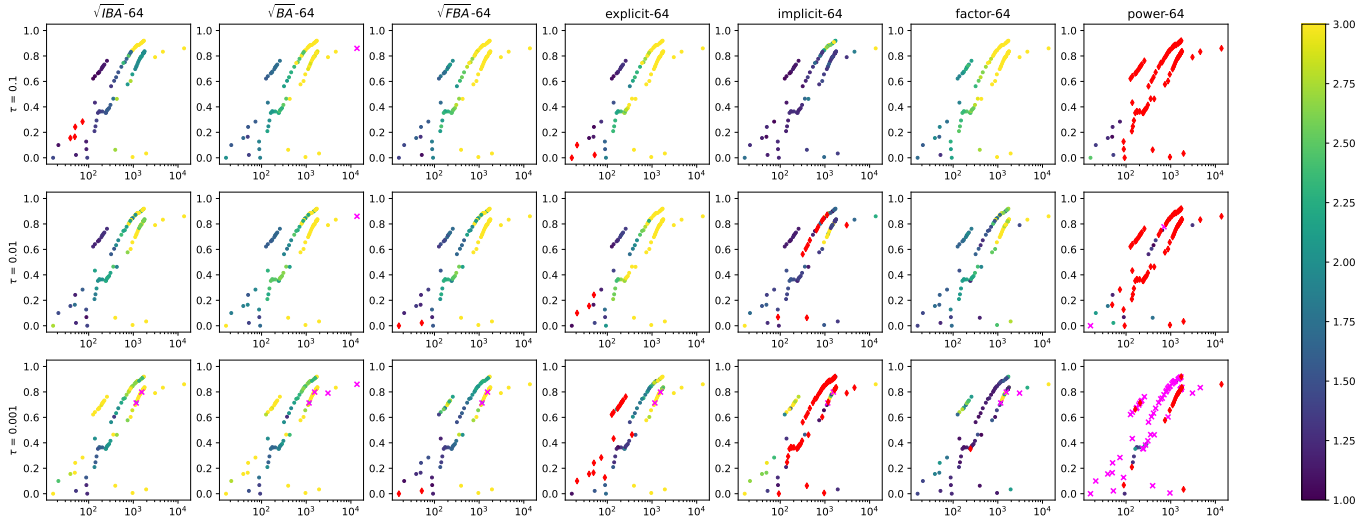


Figure 4.3: Solvers performance in double precision on BAL problems characterized by the number of cameras and RCS sparsity. Each column corresponds to a solver; the rows represent different accuracy tolerances $\tau = \{0.1, 0.01, 0.001\}$, where the plots in the same row have the same tolerance. In each plot, x-axis shows the number of cameras in log scale, y-axis shows the RCS sparsity in fraction. Each plot corresponding to the performance of a solver and an accuracy threshold, in which each problem is shown as either diamond (first to achieve the threshold), pink cross (never achieve the threshold), or a colored dot (encoding the relative time α up to 3.0).

problems, in particular at lower tolerances which require many PCG iterations. We investigate the performance difference on problems of different scale and sparsity in Section 4.5.

Factor SC has a pattern similar to both explicit and implicit SC solvers. It acts more similar to explicit SC at higher tolerance, where the performance is worse on larger problems, and it has similar performance as implicit at low tolerance. We explain this phenomena in Section 4.6.2. In short, factor SC combines the advantages from both solvers. Factor grouping improves the runtime required to build the explicit representation, and mitigate the drawback of constructing explicit representation on large-scale problems by using implicit representation. However, grouping the landmarks requires additional time, which is amortized only at high accuracy threshold

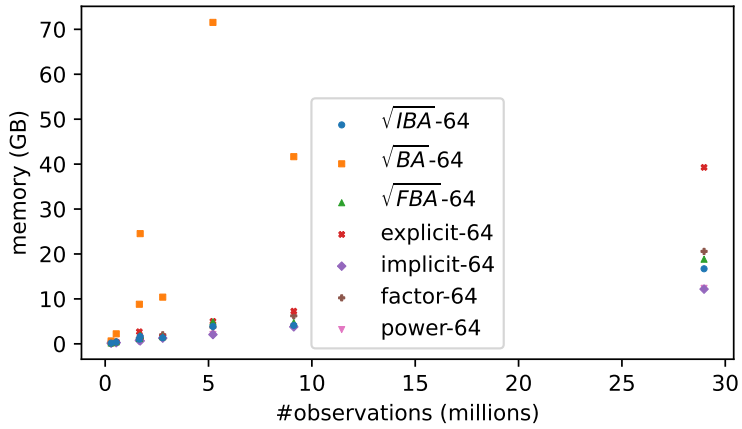


Figure 4.4: Memory usage on the BAL datasets in double precision. x-axis is the peak memory in GB, y-axis is the number of observations of the problems.

where more PCG iterations are required.

For the square root solvers, they are overall competitive on small to medium-scale problems at higher tolerances, but noticeably worse than other solvers at lower tolerance. In Section 4.7, we investigate the runtime spent on each component of the solvers. In summary, $\sqrt{BA}$ is more expensive for preparing the linear system before solving with PCG; $\sqrt{IBA}$ has a slightly more expensive multiplication but cheaper preparation compared to $\sqrt{BA}$. In Section 4.5, we show PCG often requires more iterations at lower tolerance, hence $\sqrt{IBA}$ is better than $\sqrt{BA}$ at high tolerance. Similarly, as explicit representation also has a cheaper multiplication, the performance of $\sqrt{FBA}$ becomes better at low tolerance.

Memory usage

Memory consumption is another critical aspect of a solver. In Figure 4.4, we illustrate the peak memory used by the solvers during optimization. The memory usage grow is approximately linear with the number of observations in a problem.

Except for $\sqrt{BA}$, all solvers store the Jacobian in the more compact SC landmark blocks (Figure 3.5). In contrast, $\sqrt{BA}$ stores the Jacobian in the larger $\sqrt{BA}$ landmark blocks (Figure 3.6). Landmark blocks always allocate the most amount of memory in our experiments. Since $\sqrt{BA}$ utilizes the less compact landmark block, it always consumes the most memory as evidenced by the result. There are several problems that $\sqrt{BA}$ requires significant more memory: *final961*, *final1936*, and *final4585*. The major distinction of these problems is the maximum number of observations per landmark.

Typically, the number of BAL problems is between 100 to 200, but the numbers of these problems are 839, 1293, and 535, respectively.

As the camera Jacobian blocks are stored diagonally in the $\sqrt{BA}$ landmark block, it needs significant more memory that scales quadratically with the number of observations of the landmark compared to the SC landmark block. For instance, the largest problem *final13682*, which is shown as the rightmost problem in Figure 4.3, has a maximum of 1748 observations in all landmarks. $\sqrt{BA}$ runs out of memory for allocating landmark blocks on this problem in our experiment.

To analyze the memory consumption of other solvers, we can inspect the memory usage on the largest problem, which demonstrates a more obvious difference. Both implicit SC and power BA require the lowest amount of memory. Beside the SC landmark blocks, implicit SC needs memory to allocate the block diagonal matrix $\mathbf{H}_{ll}^{-1}$ and the diagonal blocks of $\widetilde{\mathbf{H}}_{pp}$ for the preconditioner. On the other hand, power BA needs memory for $\mathbf{H}_{ll}^{-1}$ and $\mathbf{H}_{pp}^{-1}$. All of those are block diagonal matrices which require only little memory.

On the other hand, explicit SC allocates considerably more memory on the largest problem *final13682*, which is due to extra memory is required to store $\widetilde{\mathbf{H}}_{pp}$. Despite *final13682* is a sparse problem, it still scales quadratically with the number of cameras. This also stands for factor SC and $\sqrt{FBA}$, where each factor allocates a dense Hessian block. However, because the number of cameras in each block is far less than the total number of cameras of the problem, the memory use of these two solvers is considerably less affected by the quadratics complexity.

Compared to $\sqrt{BA}$ which marginalizes on the landmark block, $\sqrt{IBA}$ and $\sqrt{FBA}$ allocate a dedicated block of memory for marginalization. This requires fewer memory than $\sqrt{BA}$ as demonstrated in the result.

4.4.2 Experiments on the 1DSfM dataset

In Figure 4.5, we demonstrate the performance profiles evaluated on the 1DSfM problems in double precision, where the result is generally similar to the BAL dataset with a few exceptions.

As previously stated, 1DSfM dataset comprises of mostly small problems. Compared to the result on BAL, $\sqrt{FBA}$, explicit SC, and factor SC solver are substantially benefited from the explicit representation. Furthermore, implicit SC maintains good accuracy and speed, which explains why it is the de facto choice for bundle adjustment solvers.

On the other hand, the accuracy of power BA considerably drops compared to the BAL result, particularly at lower tolerances. As previously mentioned, 1DSfM problems are often more difficult to solve, which is reflected by the higher number of iterations of the other solvers. As demonstrated in Table 4.4, the final cost of power BA on average is

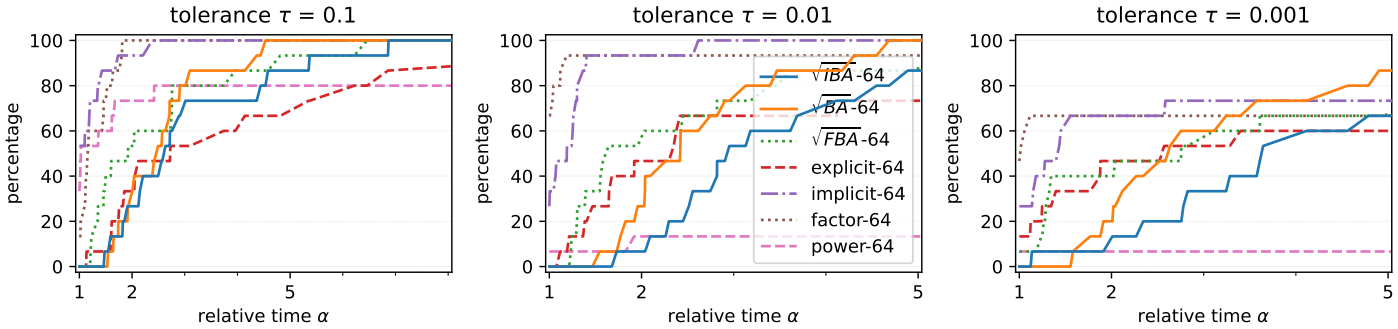


Figure 4.5: Performance profiles evaluated on 1DSfM problems in double precision.

more than the cost of other solvers, indicating that power BA has insufficient order for estimating an accurate solution. This emphasizes the significance of a good termination criteria for a solver.

	$\sqrt{IBA}$-64	$\sqrt{BA}$-64	$\sqrt{FBA}$-64	explicit-64	implicit-64	factor-64	power-64
final cost (relative)	**0.999**	**0.999**	1.001	*1.000*	*1.000*	1.004	1.189
time (relative)	2.969	2.236	1.828	2.057	1.000	*0.820*	**0.456**
#Iterations	511	531	*464*	488	481	484	**198**

Table 4.4: Solvers summary of 1DSfM result. It shows the average relative runtime, the relative final cost, and the total number of inner iterations. For power BA solvers, the inner iterations refer to the approximation order, while for other solvers, inner iterations refer to the number of PCG iterations.

4.4.3 Experiments on the MCBA dataset

MCBA primarily comprises of large-scale and dense problems with a high number of observations per landmark, therefore it places an emphasis on the efficiency of a solver. It is used to evaluate our most runtime and memory efficient solvers, $\sqrt{IBA}$, implicit SC, and power BA. Given the scale of the problems, other solvers would take significant more time to optimize.

The performance profiles run in double precision are shown in Figure 4.6. At high tolerance $\tau = 0.1$, power BA is approximately 50% to 100% faster than implicit SC on half of the problems. However, power BA is unable to achieve the tolerance on more of

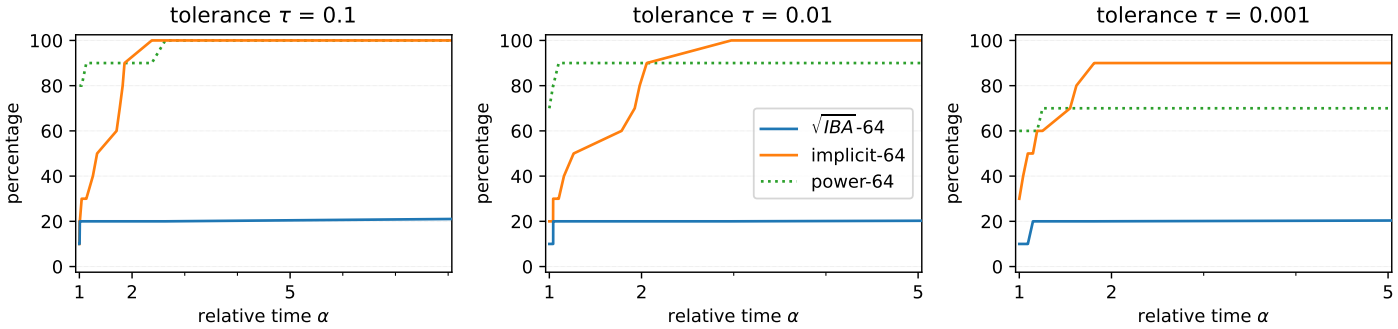


Figure 4.6: Performance profiles evaluated on MCBA problems in double precision.

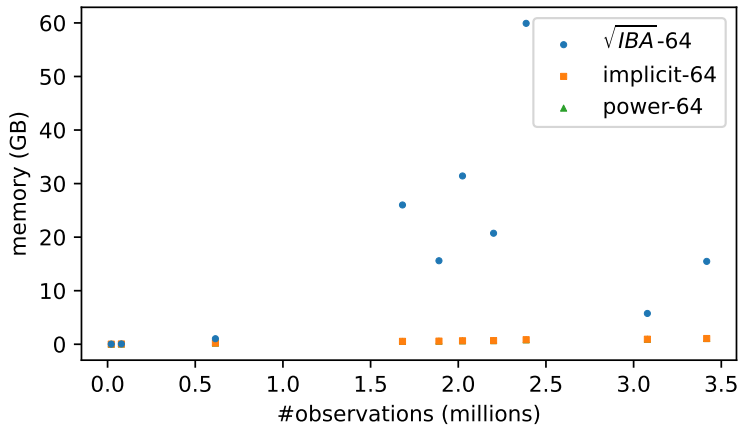
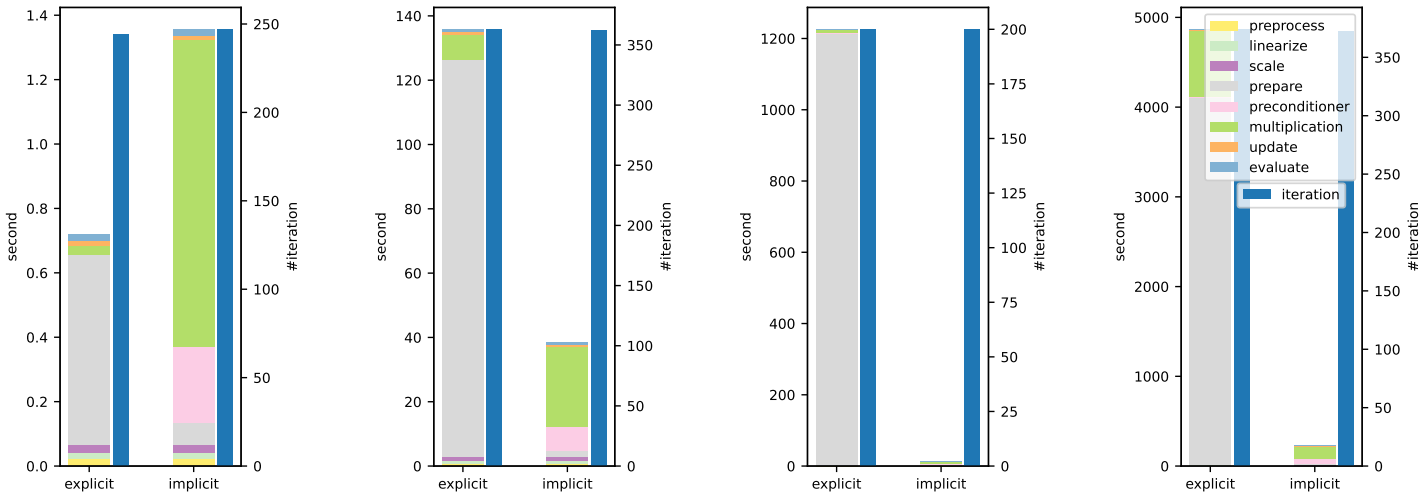


Figure 4.7: Memory usage on MCBA datasets in double precision.

problems than implicit SC does at low tolerance $\tau = 0.001$. It once again indicates that the currently used order of power BA is insufficient to obtain an accurate result. At higher tolerances $\tau = \{0.1, 0.01\}$, power BA is up to two times faster than implicit SC. In Section 4.9, we show that power BA has a better convergence rate than implicit SC.

Across all tolerances, $\sqrt{IBA}$ solve only two small problems. While $\sqrt{IBA}$ is capable of solving all problems without running out of memory, it is unable to solve the large-scale problems even to the highest tolerance in the given time. In the result, it takes substantially longer to solve the large-scale problems, which is why it is not depicted in the figure.

Compared to BAL problems, MCBA problems have a much higher high maximum number of observations per landmark. As a result, $\sqrt{IBA}$ requires significantly more



(a) trafalgar50 (small-scale)

(b) venice1184 (medium-scale)

(c) final961 (dense)

(d) final13682 (sparse, large-scale)

Figure 4.8: Runtime of each individual phase of explicit and implicit SC solver on four BAL problems with different characteristics evaluated in double precision. In each plot, a solver is represented by two bars. The left stacked bar shows the total runtime of the individual phases spent in all 20 LM iterations (left axis); the right bar indicates the total number of PCG iterations (right axis).

memory and runtime for marginalization. Figure 4.7 shows the memory consumption of the evaluated solvers. This result again demonstrates that square root solvers are less advantageous for large-scale problems.

4.5 Comparison of Explicit and Implicit SC Solver

4.5.1 Performance Analysis

We demonstrate that the explicit SC solver outperforms the implicit SC solver on small to medium-sized problems in Section 4.4.1. To investigate the properties of two solvers in detail, we break each solver down into its different phases, and then measure the runtime spent on each phase. We categorize the computation into the following 7 phases:

preprocess Allocate SC landmark block.

linearize Linearize the problem (2.30), then store the Jacobians in the landmark blocks.

scale Scale the Jacobians as described in Section 2.2.1.

prepare Prepare the linear system for solving via PCG, e.g. prepare $\tilde{\mathbf{H}}_{pp}$ and $\widetilde{\mathbf{b}}_p$.

preconditioner Compute the preconditioner, i.e. the diagonal blocks of $\widetilde{\mathbf{H}}_{pp}^{-1}$.

multiplication Solve the reduced system via PCG.

update Update the landmarks and camera parameters according to the solution.

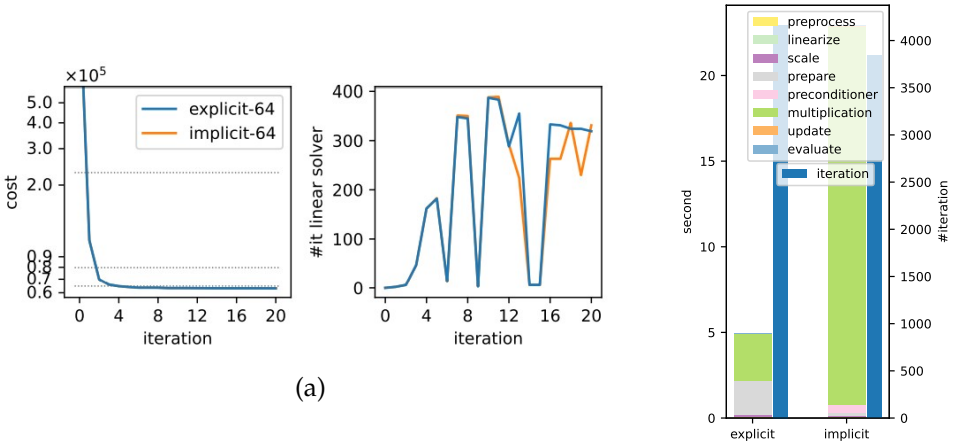
evaluate Evaluate the actual cost reduction with the updated parameters.

We present the runtime of both solvers on four distinct problems in Figure 4.8. In the result, the runtime of **prepare**, **preconditioner**, and **multiplication** vary greatly in between problems. The reason is that the solvers perform different computations in each phase. Apart from computing $\widetilde{\mathbf{b}}_p$ during **prepare**, the explicit SC solver constructs the sparse block matrix $\tilde{\mathbf{H}}_{pp}$, whereas implicit SC simply constructs the diagonal matrix $\mathbf{H}_{ll}^{-1}$, which involves less computation. This is reflected in the result, as explicit SC spends significantly more time than implicit SC on all problems during **prepare**.

In the result, **prepare** consumes the most time for explicit SC, which is because building $\tilde{\mathbf{H}}_{pp}$ explicitly is computationally inefficient. Given a landmark with n observations, to compute the sub-blocks of $\tilde{\mathbf{H}}_{pp}$ associated to the landmark, we need to build a double for loop that iterates the camera Jacobian blocks $\mathbb{R}^{2\times 9}$ in the landmark block, which has a time complexity of $O(n^2)$. Implicit SC, on the other hand, only needs to compute $\mathbf{H}_{ll}^{-1} \in \mathbb{R}^{3\times 3}$ with $J_l^\top J_l$ for each landmark, which has a time complexity of $O(n)$. As a result, explicit SC spends substantially more time on **prepare** than implicit SC. This is also indicated in Figure 4.8, where explicit SC performs particularly worse on dense and larger problems.

Implicit SC takes significantly longer than explicit SC for **preconditioner**. While implicit SC needs to build the diagonal blocks of $\widetilde{\mathbf{H}}_{pp}$ then inverting them, explicit SC just has to computes $\widetilde{\mathbf{H}}_{pp}^{-1}$ by inverting the diagonal blocks of $\widetilde{\mathbf{H}}_{pp}$ computed in the previous phase. Nonetheless, the time complexity of implicit SC only scales with the number of observations $O(N_r)$, which is still overall inexpensive.

For **multiplication**, as implicit SC involves more computations to multiply with the sub-blocks of Jacobians per PCG iteration (3.2), explicit SC typically outperforms implicit SC. However, explicit SC is less advantageous on larger problems demonstrated in Figure 4.8, where **multiplication** becomes expensive on problems with increasing



(a)

(b)

Figure 4.9: Performance of explicit and implicit SC solver on BAL trafalgar215 in double precision. The cost and the number of PCG iteration at every LM iteration is shown in (a). The gray dashed lines indicate the cost of difference tolerance $\tau = \{0.1, 0.01, 0.001\}$. The total runtime of individual solvers are shown in (b).

number of cameras. On the smaller problems (Figure 4.8a and 4.8b), **multiplication** of explicit SC is faster than implicit SC, but slower on large problem shown in Figure 4.8d.

In summary, explicit SC has a quadratic complexity in terms of constructing $\widetilde{\mathbf{H}}_{pp}$ matrix, yet multiplication with the matrix is rather cheap. As a result, explicit SC is faster on larger problems than on smaller problems due to the quadratic time complexity. The operations of implicit SC has a linear complexity. on larger problems, it typically outperforms explicit SC, and it is especially advantageous on large-scale dense problems.

4.5.2 Tolerance

Previously, we present the performance of our solvers in Figure 4.3, it demonstrates a tendency toward improving explicit SC performance with lower tolerances. We show the reason is because PCG requires more iterations to approximate the solution at low tolerance by Figure 4.9.

Figure 4.9a illustrates the convergence state of both solvers. On the left is the cost

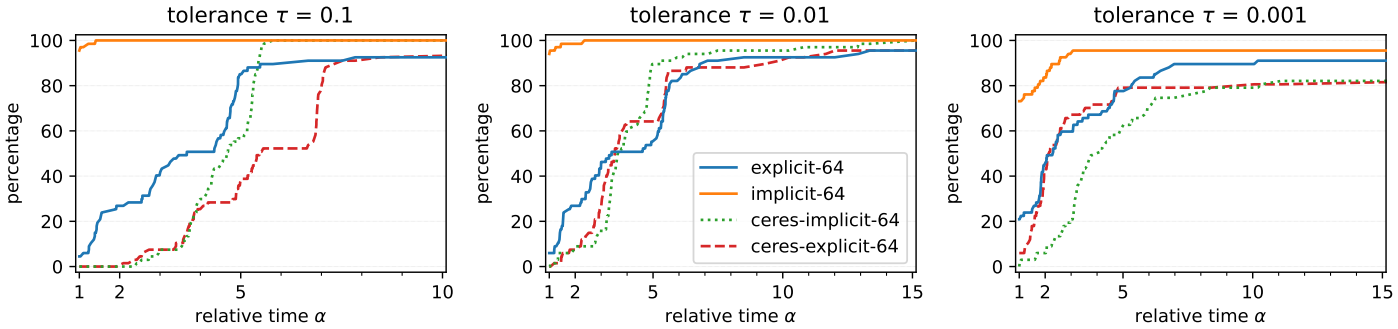


Figure 4.10: Performance profiles of Ceres and our implementation of explicit and implicit SC, evaluated on BAL problems in double precision.

at each LM iteration, and on the right is the number of PCG iterations at each LM iteration. Both solvers achieve tolerance $\tau = 0.01$ within the first few iterations, and the remaining LM iterations try the reduce the cost to the lowest tolerance $\tau = 0.001$. After achieving tolerance $\tau = 0.01$, both solvers conduct more PCG iterations compared to the prior LM iteration.

Figure 4.9b demonstrates the runtime spent on each phase. Despite the expensive **prepare**, explicit SC overall outperforms implicit SC with its much cheaper **multiplication**. The reason is because both solvers have a higher number of PCG iterations on this problem, which allows the slow **prepare** to be traded off for the faster **multiplication**.

While implicit SC consistently beats explicit SC on medium- to large-scale problems, the performance of the explicit SC solver is ultimately a trade-off between **prepare** and **multiplication**, which is dependent on the characteristics of the problem. To summarize, explicit SC can perform well in the following situations:

- Solving Small-scale problems, where the advantage of cheap **multiplication** outperforms the expensive **prepare**.
- Solving small- to medium-scale problems at lower tolerances, where the negative effect of **prepare** is less prominent and the efficiency of **multiplication** becomes crucial due to the high number of PCG iterations.

4.5.3 Performance comparison with Ceres

Ceres solver [AMT22] is a popular open source C++ library for solving non-linear least squares problems. With its excellent code quality and speed, it is widely used in computer vision and robotics areas. Although Ceres can be used a general-purpose

solver, it is also specially designed for large-scale BA problems, which is an interesting baseline to compare with our implementation. Ceres can solve a BA problem with PCG also providing the options to solve explicitly (ceres-explicit) or implicitly (ceres-implicit). We compile Ceres with Eigen and multithreading support. Our results are evaluated using the same configuration as in [Dem+21], which closely matches the our custom solver setting for fair comparison.

We evaluate the performance of Ceres and our implementation of explicit and implicit SC in Figure 4.10. Across all tolerances, our implicit SC implementation outperformed other solvers by a margin. With higher tolerances $\tau = \{0.1, 0.01\}$, our implicit SC solver was around 4 to 5 times faster than Ceres implicit SC solver. Likewise, our explicit SC solver was also faster than Ceres at high tolerance $\tau = 0.1$, and had a similar at lower tolerances $\tau = \{0.01, 0.001\}$.

On the other hand, Ceres' explicit and implicit SC implementations eventually achieved the same accuracy as our approach, but took significantly longer time. At low tolerance $\tau = 0.001$, Ceres implementations do not accomplish the same level of accuracy in the plot, indicating that Ceres requires substantially more time to obtain the same level of accuracy.

Our implementation is optimized for bundle adjustment and is built leveraging highly parallelized computation and vectorized operations. Additionally, the adoption of the SC landmark block enables more efficient memory consumption and a more predictable memory access pattern. While Ceres makes use of multithreading, its general-purpose implicit SC solver, in particular, cannot take advantage of parallelism to the extent that our custom can. Given Ceres is a general purpose solver, the superior performance of our implementation demonstrates the importance of custom implementation for specific problems.

4.6 Factor SC Solver

In [Car+14], factor grouping is proposed to apply on BA problems to reduce the runtime of multiplications during PCG. It aims to improve the time complexity of the vector-matrix multiplication with RCS by grouping landmarks and selecting the best representation for each group. The authors validate the improvement on a subset of BAL problems, which are all small- to medium-scale. In this section, we study each component of factor SC solver in detail, and evaluate its performance on a variety of problems using multiple visualizations.

To begin, we study the grouping results and performance of FP-tree. Then, we evaluate the performance of factor SC on the entire BAL dataset. It is compared to both explicit and implicit SC since it is a hybrid of the two. Following that, we study

the benefits of factor grouping by examining the runtime of its individual phases. Lastly, we compare the performance of our factor SC implementation to the result demonstrated in the paper.

4.6.1 FP-tree

problem	grouping time (s)	#factors	#grouped landmarks	#implicit landmarks
dubrovnik150	0.2	2656	53502	42319
dubrovnik182	0.23	3122	73516	43254
ladybug1118	0.2	4733	86154	32230
ladybug1469	0.25	5895	106869	38330
trafalgar170	0.06	1652	35988	13279
trafalgar215	0.08	1841	42195	13715
final4585	3.3	39576	767695	556887
final13682	11.41	155494	2572637	1883480

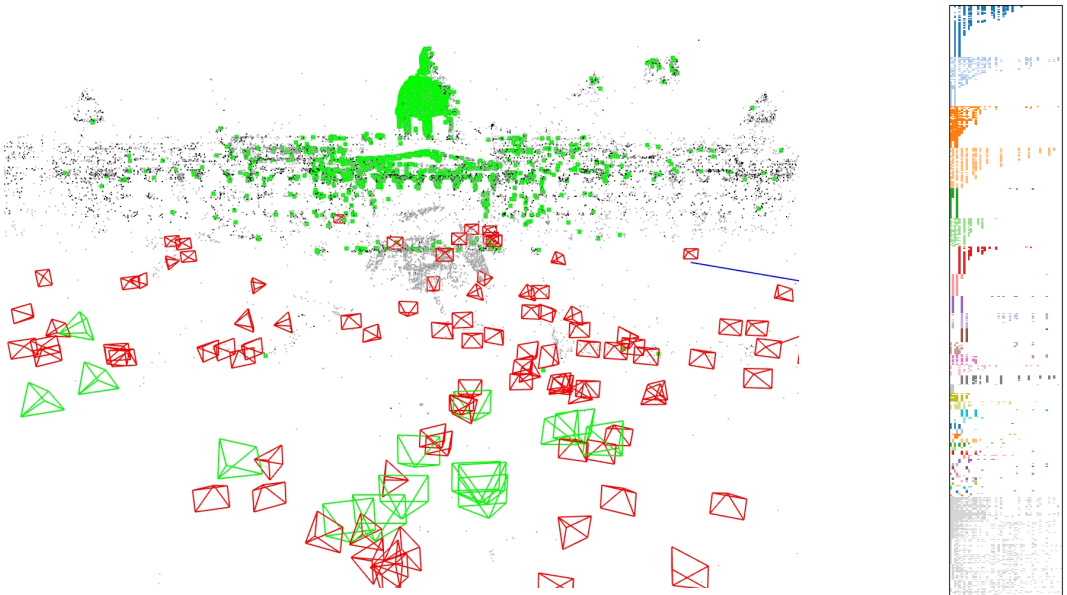
Table 4.5: Performance of our FP-tree implementation and statistic on selected BAL problems. The table shows the time to group factors, number of factors, number of grouped landmarks, and the number of implicit landmarks (landmarks which do not belong to any factor and use implicit representation).

We reimplement the FP-tree described in the paper [Car+14], and we parallelize the tree operations as much as possible in our implementation. Given that this is a different implementation from the one in the paper, it is interesting to compare the result. But note that the hardware on which we conduct the experiments should be far recent than the hardware used in the paper.

In Table 4.5, we evaluate our FP-tree implementation on the same set of problems in the paper, as well as 2 large-scale problems. Our implementation is faster than the result demonstrated in the paper. Furthermore, the number of factors and grouped landmarks are slightly different than the ones in the paper. Given the slightly faster grouping runtime and comparable grouping result, our factor SC solver should demonstrate similar performance shown in the paper.

In Figure 4.11, we visualize the grouping result by our FP-tree. Figure 4.11a shows the landmarks and camera poses correspond to one of the factor groups. Landmarks and cameras in the group are highlighted in green. As previously described, FP-tree groups landmarks that are convisible by the same set of cameras, and the number of landmarks is always greater the number of cameras in a factor group.

In Figure 4.11b, we illustrate the structure of a small problem. Each row represents a landmark, and each column represents a camera. The cameras observing each landmark



(a) (b)

Figure 4.11: Visualizations of the FP-tree grouping result. Figure (a) shows the cameras poses (polygon) and landmarks (points) of BAL trafalgar257, where the cameras and landmarks belonging to the largest factor are indicated in green. Figure (b) shows the structure of a reduced BAL ladybug49 problem, where each row encodes a landmark. The landmarks do not belong to any factor group are indicated by gray at the bottom, and landmarks belonging to the same group are indicated by the same color.

are color-coded in the corresponding row. The landmarks which do not belong to any factor group are indicated by gray at the bottom, and the landmarks belonging to the same factor group are placed together and indicated by the same color.

At the bottom, we can notice the landmarks which do not correspond to a group have considerably more observations compared to the grouped landmarks. This is because landmarks with many camera observations are more difficult to group than those with only a few observations. Additionally, they are more challenging to merge into existing factor groups, as their camera observations are rarely a subset of any group. As a result, implicitly represented landmarks typically have more observations than explicitly represented landmarks.

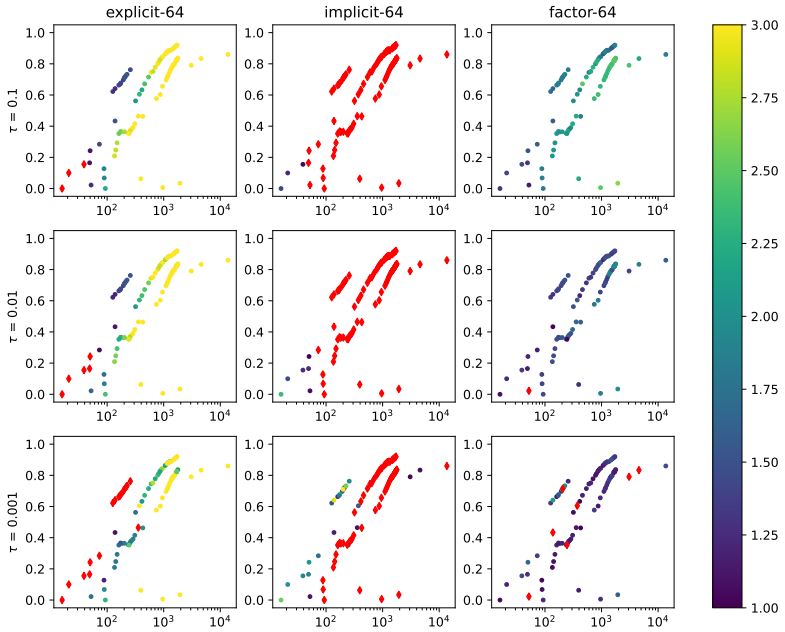
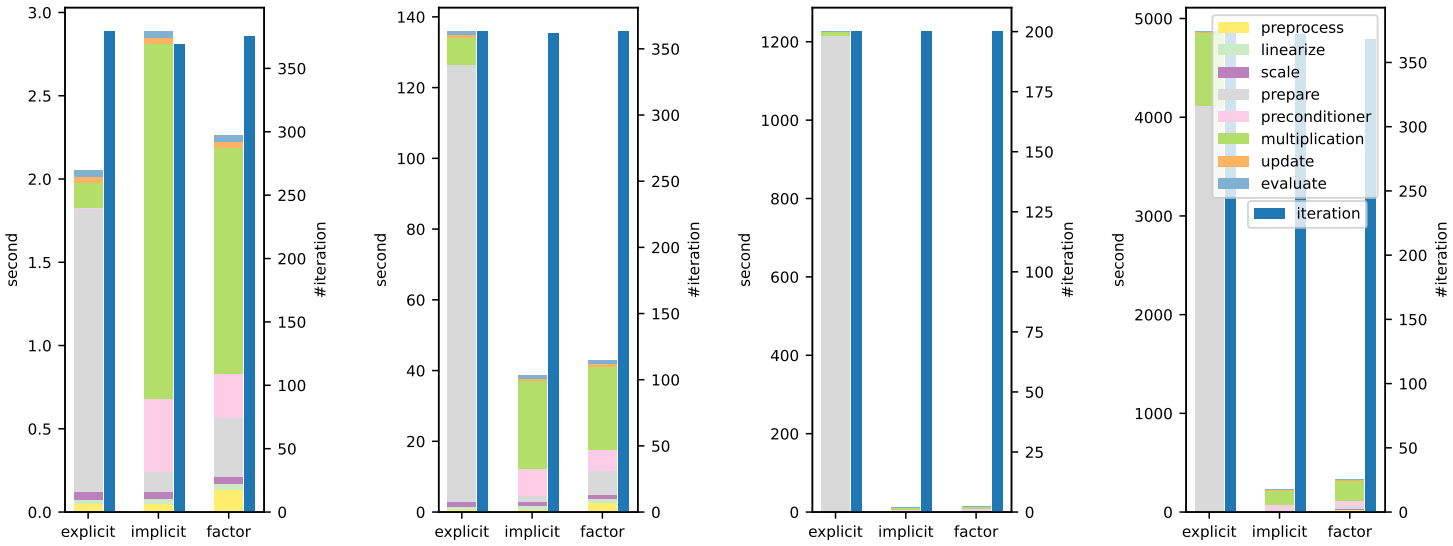


Figure 4.12: Performance of explicit, implicit, and factor SC evaluated in double precision on BAL problems characterized by the number of cameras and RCS sparsity.

4.6.2 Performance analysis

Figure 4.12 shows the performance comparison of the explicit, implicit, and factor SC solvers evaluated on BAL dataset. Across all tolerances, we can see factor SC solver is clearly a blend of explicit and implicit SC. In Section 4.5, we conclude that explicit SC performs better on smaller problems but worse on larger problems compared to implicit SC, and factor SC performs competitive on both small- and large-scale problems. In other words, factor SC solver achieves a much more stable performance on varied kinds of problems, it consistently scores close to the fastest solver on every problems, particularly at lower tolerances $\tau = \{0.01, 0.001\}$, where the overhead to compute the factor grouping can be better amortized.

Figure 4.13 shows the runtime evaluated on four distinct problems solved with 20 LM iterations. Alongside with allocation time, **preprocess** now also includes time for factor grouping by our FP-tree. As illustrated in Figure 4.13a and 4.13b, factor grouping



(a) trafalgar161 (small-scale)

(b) venice1184 (medium-scale)

(c) final961 (dense)

(d) final13682 (sparse,large-scale)

Figure 4.13: Runtime comparison of explicit, implicit, and factor SC solver on four BAL problems with distinct characteristics evaluated in double precision.

consumes very little extra runtime, which only has a small impact on solver overall performance. Note that the figure shows the total runtime, hence it primarily reflects the performance at low tolerance. Since factor grouping is performed once before the optimization, the impact on performance will be more noticeable if a solver runs only a few iterations.

Across all problems in Figure 4.13, the runtime spent in the **prepare** phase of factor SC is significantly reduced compared to explicit SC. In Section 4.5, we conclude that constructing $\tilde{\mathbf{H}}_{pp}$ explicitly has a quadratic time complexity, and hence **prepare** phase of explicit SC consumes an extraordinary amount of runtime on dense or larger problems, demonstrated again in Figure 4.13.

Despite the factor SC solver also constructs the explicit representation of the RCS for some landmarks, factor grouping aids in mitigating the impact of quadratic complexity. As illustrated in Figure 4.11b, the landmarks which have more observations mostly do not belong to any factor group, therefore those landmarks are expressed as implicit representation, which has a linear complexity. As a result, we are able to avoid building

the costly explicit representation of those landmarks with many observations. On dense or large-scale problems, as illustrated in Figure 4.13c and 4.13d, the runtime of **prepare** is significantly improved compared to explicit SC.

Furthermore, the runtime spent in **preconditioner** of factor SC is also noticeable improved. As shown in Figure 4.11b, the majority of landmarks are grouped, thereby expressing as explicit representation. Consequently, the **preconditioner** phase of factor SC is faster than implicit SC because building the preconditioner with the explicit representation is relatively cheaper.

In Section 4.5.1, we reveal that multiplication with the explicit representation is often slower on larger problems, which holds true for the factor SC solver as well. In Figure 4.13, the **multiplication** phase of factor SC also becomes less advantages on problems with increasing scale compared to implicit SC.

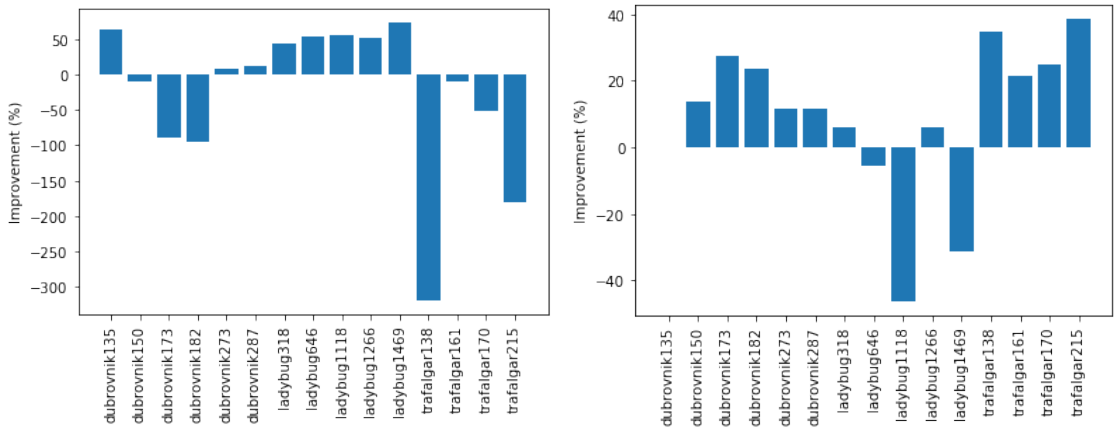
Similarly, because factor SC incorporates the implicit representation as well, the advantage of cheaper multiplication with explicit representation is diminished compared to explicit SC on smaller problems. As shown in Figure 4.13a, **multiplication** is a substantial fraction of the total runtime.

On the other hand, the shortcoming of both representations are mitigated by factor grouping. Compared to the explicit SC, **multiplication** phase of factor SC constitutes only a small factor of the total runtime on large problems demonstrated by Figure 4.13d. In Section 4.5.1, we conclude that explicitly constructing RCS matrix $\widetilde{\mathbf{H}}_{pp}$ has a quadratic complexity in the number of cameras. With factor grouping, the number of cameras in each factor group is significantly lower than the total number of cameras of the problem. As a result, a lower camera count in each group minimizes the impact of quadratic complexity when constructing the explicit representation, which prevents the explicit representation severely hindering the performance on larger problems.

In the results shown in Figure 4.12, the explicit and the implicit SC solver only perform competitively on small- or large- scale problems. By applying factor grouping, factor SC now produces competitive result across problems with different scale and sparsity. This demonstrates that factor grouping allow factor SC overcoming the weakness of explicit and implicit SC solvers while absorbing their strengths.

However, as our results demonstrate, factor SC is often not the fastest solver, where it is slower on smaller issues than explicit SC and slower on larger problems than implicit SC as demonstrated in Figure 4.12. This is a downside of combining two solvers. As one solver is often faster than the other on certain problems, when the two solvers are combined, the benefits of using the faster solver will be slightly compromised. However, factor SC becomes an all-round solver that is competitive on all types of problems, it has consistent performance on problems with different characteristics across all tolerances.

According to our findings, we suggest factor SC for solving small- to medium-scale problems because it performs consistently regardless of the sparsity of a problem. It



(a) Explicit SC

(b) Implicit SC

Figure 4.14: Total runtime improvement of factor SC over explicit SC (a) and implicit SC (b) on the same set of problems in [Car+14], solved with 20 LM iterations and evaluated in double precision. Let t_f and t as the total runtime of factor SC and the comparing solver on a problem respectively, then the runtime improvement using factor grouping is computed as $(t - t_f)/t$.

is especially recommended if a precise solution is desired, as factor grouping is more advantageous at lower tolerances.

4.6.3 Performance Comparison with the Original Implementation

Given that we reimplement the factor grouping method in our framework, it is interesting to compare our implementation to the performance of the one in the paper. We evaluated the explicit, implicit, and factor SC solvers on the same set of problems, which are small- to medium-scale problems. Because the number of PCG iterations varies across solvers due to numerical instability, it is better to compare the general pattern rather than particular problem in our result.

We compare the performance of factor SC to explicit and implicit SC in Figure 4.14. When the bar of a problem is positive, it indicates factor SC outperform the corresponding solver on that problem, and vice versa. The solvers are evaluated on a subset of BAL problems, which include small-scale problems from dubrovnik and trafalgar, and medium-scale problems from ladybug.

In Figure 4.14a, the performance of factor SC is compared to explicit SC. Factor SC overall performs worse on small-scale problems and outperforms explicit SC only on medium-scale ladybug problems. In comparison to implicit SC, factor SC is better at

solving small-scale problems but falls short at solving medium-scale problems in Figure 4.14b. Our result coincides with our previous conclusion regarding the performance of factor SC.

However, the implementation of factor SC in the paper outperforms implicit SC on all the selected problems, while our implementation outperforms implicit SC only on small-scale problems. Given the disparity on the medium-scale ladybug problems, it is possible that there could be major differences between our and the authors' factor SC implementations. Either their implementation of explicit representation is significantly faster on larger problems, or our implicit SC implementation is faster on smaller problems.

In additional to the paper, we demonstrate that the time complexity of constructing RCS representations is as critical as the complexity of PCG multiplication. We thoroughly evaluate the performance of factor SC on the full BAL dataset. Compared to implicit SC, factor grouping only demonstrates improvement on smaller problems while not on the larger problems in our result.

4.7 Square Root Solvers

In this section, we compare the performance of the $\sqrt{BA}$, $\sqrt{IBA}$, and $\sqrt{FBA}$ solvers. First, we evaluate the performance of the three solvers on the BAL dataset. Then, we analyze the runtime of the individual phases of the solvers. Finally, we discuss the impact of memory consumption on runtime.

4.7.1 Performance Analysis

We demonstrate the performance difference of the square root solvers on the BAL dataset in Figure 4.15. In short, $\sqrt{IBA}$ is overall the fastest solver at high tolerance, while $\sqrt{FBA}$ is faster at low tolerance, and the performance of $\sqrt{BA}$ is a blend of these two solvers. On average, the performance difference between the square root solvers is relatively small on varied problems compared to SC solvers.

Since $\sqrt{IBA}$ repeatedly marginalizes landmarks in every PCG iteration, $\sqrt{IBA}$ conducts more floating point operations than $\sqrt{BA}$, therefore $\sqrt{IBA}$ should be slower than $\sqrt{BA}$. But in practice, $\sqrt{IBA}$ demonstrates overall better performance at higher tolerances $\tau = \{0.1, 0.01\}$ and similar performance at low tolerance $\tau = 0.001$.

For $\sqrt{FBA}$, as it uses the explicit RCS representation for some of the landmarks, the performance shows a similar trend as factor SC solver, where it becomes relatively faster at lower tolerances. As a result, it is the fastest solver at tolerance $\tau = 0.001$. It indicates multiplication with the explicit RCS matrix is faster than multiplying two times with $\mathbf{Q}_2^\top \mathbf{J}_p$.

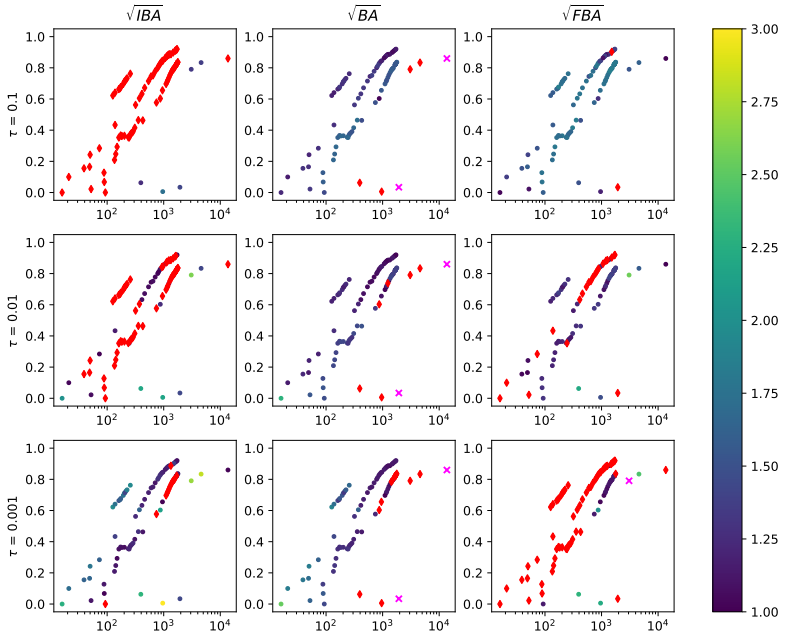


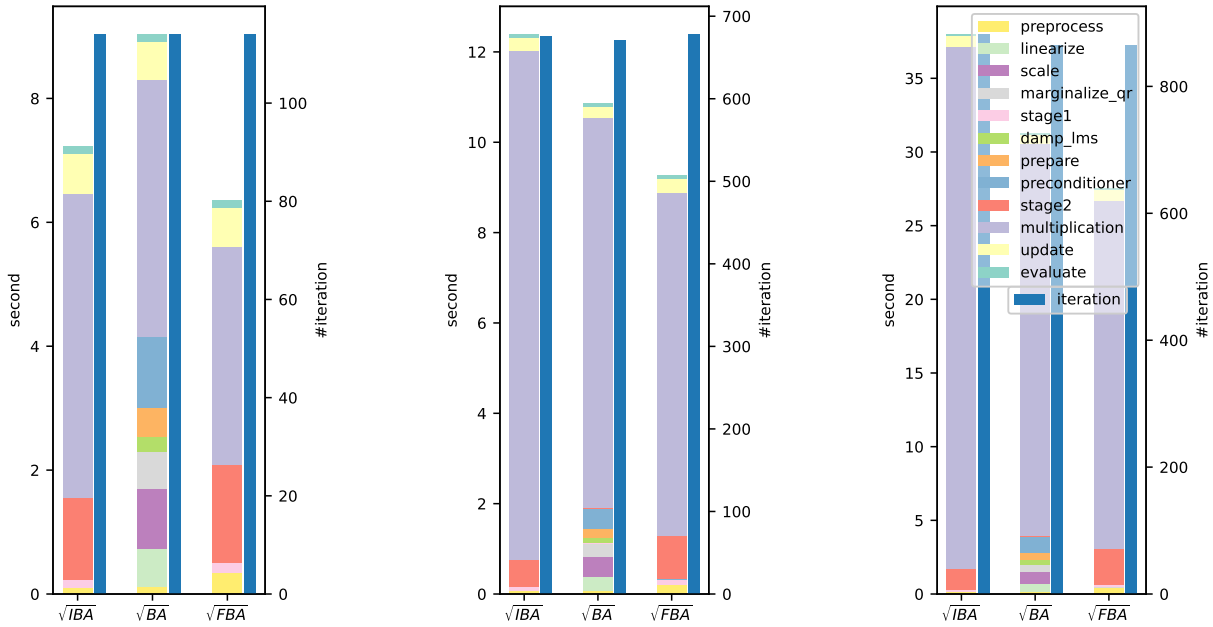
Figure 4.15: Performance of square root solvers evaluated in double precision on BAL problems.

In Figure 4.16, we show the runtime of individual phases of the square root solvers. In comparsion to SC solvers, square root solvers involve two additional phases: marginalizing landmarks using QR decomposition and damping the landmark variables. They are denoted as **marginalize_qr** and **damp_lms** in the figure.

Because $\sqrt{IBA}$ and $\sqrt{FBA}$ must copy the Jacobian to a dedicated memory block and then execute multiple phases sequentially without interruption, we group the individual phases and categorize them **stage1** and **stage2**. **stage1** includes **linearize** and scaling J_l in the landmark blocks. **stage2** includes scaling J_p, **damp_lms**, **marginalize_qr**, **prepare**, and **preconditioner**. In our study, we treat **stage1** and **stage2** as a whole.

Furthermore, as QR decomposition is performed once per PCG iteration at the landmark blocks for $\sqrt{IBA}$ and $\sqrt{FBA}$, the operations performed during **multiplication** are slightly different between square root solvers.

Across all problems in our result, the $\sqrt{BA}$ phases before solving the linear system contribute a significant amount of the total runtime. By comparison, **stage1** and **stage2** of other two solvers only consume around half the time that $\sqrt{BA}$ does. In our result



(a) dubrovnik135 (small-scale)

(b) ladybug539 (medium-scale)

(c) ladybug1514 (large-scale)

Figure 4.16: Runtime of individual phases of $\sqrt{IBA}$, $\sqrt{BA}$, and $\sqrt{FBA}$ solvers evaluated in double precision.

shown in Figure 4.16a, solely the sum of **linearize** and **scale** of $\sqrt{BA}$ surprisingly surpasses the runtime of **stage1** and **stage2** of $\sqrt{IBA}$ combined.

As illustrated in Figure 4.16a, solely the sum of **linearize** and **scale** of $\sqrt{BA}$ surprisingly surpasses the runtime of **stage1** and **stage2** of $\sqrt{IBA}$ combined. Particularly, despite the fact that **stage1** of $\sqrt{IBA}$ does more computations than **linearize** of $\sqrt{BA}$, **stage1** is still substantially faster than **linearize**. Furthermore, since **multiplication** of $\sqrt{IBA}$ performs marginalization once per PCG iterations, one would expect it takes up significantly more time compared to $\sqrt{BA}$. However, in our results, it is only more slightly expensive for $\sqrt{BA}$.

We believe that this is because $\sqrt{IBA}$ has a more efficient memory access pattern than $\sqrt{BA}$. Compared to $\sqrt{BA}$, $\sqrt{IBA}$ utilizes a more compact SC landmark block. $\sqrt{BA}$ stores the camera Jacobian blocks diagonally within the $\sqrt{BA}$ landmark blocks, whereas $\sqrt{IBA}$ stores the Jacobian block vertically within SC landmark blocks, where the blocks are placed closer in the SC landmark block than $\sqrt{BA}$.

Since we access the Jacobian block by block within a landmark block, the target

memory location of the operations is then close to each other, which leads to a better cache spatial locality. As a result, **linearize** using the SC landmark block should have fewer write misses than the $\sqrt{BA}$ landmark block, which results in a shorter runtime despite the same number of floating point operations.

Similarly, **multiplication** of $\sqrt{IBA}$ is only slightly more expensive than $\sqrt{BA}$ across the selected problems. As $\sqrt{BA}$ marginalizes on each $\sqrt{BA}$ landmark blocks, the CPU needs to recall a large amount of memory to the cache. In contrast, $\sqrt{IBA}$ copies the Jacobian blocks from the compact SC landmark block to the dedicated memory blocks, then marginalizes the landmarks on the memory blocks. As a result, most operations of $\sqrt{IBA}$ are performed on the dedicated memory blocks during optimization, which leads to a better temporal locality than $\sqrt{BA}$.

Since the number of floating point operations of RCS multiplication (3.17) is the same for $\sqrt{IBA}$ and $\sqrt{BA}$, it implies marginalizing landmarks is only slightly most costly than recalling the $\sqrt{BA}$ landmark blocks from memory as demonstrated in Figure 4.16. This highlights the importance of memory locality on the performance of a solver.

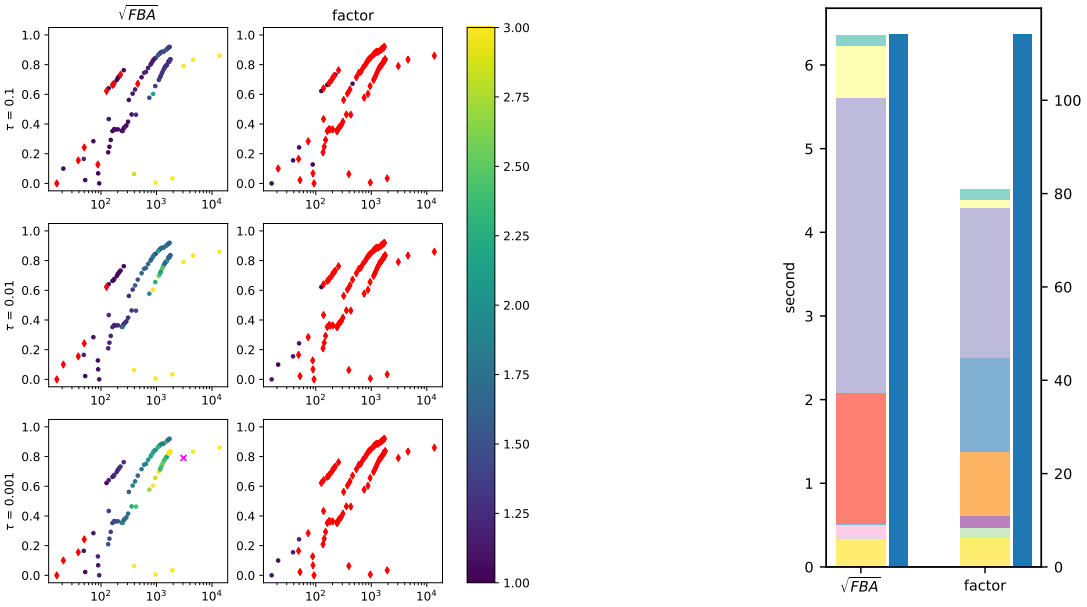
Moreover, since **stage1** and **stage2** group multiple phases together, the landmark blocks of $\sqrt{IBA}$ and $\sqrt{FBA}$ are recalled less frequently compared to $\sqrt{BA}$, where it recalls all landmark blocks once for each phase. Consequently, $\sqrt{IBA}$ and $\sqrt{FBA}$ have a more efficient memory pattern than the corresponding phases of $\sqrt{BA}$, demonstrated by the faster runtime in our result.

In Figure 4.15 at low tolerance and Figure 4.16, the **multiplication** phase of $\sqrt{FBA}$ is often faster than the other two square root solvers. Similar to factor SC, it again benefits from the cheaper explicit RCS multiplication.

We also compare the performance of $\sqrt{FBA}$ and factor SC in Figure 4.17a. In the result, factor SC solver in general outperforms $\sqrt{FBA}$ across all tolerances, while $\sqrt{FBA}$ only achieves competitive runtime on small problems. For a $\sqrt{BA}$ landmark block, the size scales quadratically to the number of observations of the landmark. Due to the fact that multiplication is performed using $\sqrt{IBA}$ landmark blocks, both the number of floating point operations and the amount of memory to be recalled are increased in comparison to the factor SC solver.

For a $\sqrt{BA}$ landmark block, the size and floating point operations scale quadratically to the number of observations of the landmark. This is also applied to $\sqrt{FBA}$, therefore the amount of memory to be recalled and the number of floating point operations are jointly increased. Consequently, **multiplication** of $\sqrt{FBA}$ is more expensive than factor SC as illustrated in Figure 4.17b.

Given that factor grouping tends to group landmarks with fewer observations, landmarks with more observations are processed similarly to how $\sqrt{IBA}$ works in the $\sqrt{FBA}$ solver. As previously mentioned, the number of floating point operations scales quadratically to the number of observations, this leads to a worse **multiplication**



(a) (b)

Figure 4.17: Comparison of $\sqrt{FBA}$ and factor SC solvers in double precision on BAL problems is shown in (a). Runtime of individual phases on dubrovnik135 is illustrated in (b).

speed as shown in Figure 4.17b. As demonstrated by our results, factor grouping is less suitable for square root formulation than the Schur complement trick.

4.7.2 Numerical Properties

As mentioned in the original paper, the square root formulation is more numerically stable than the Schur complement. In Figure 4.18, we demonstrate the numerical stability of both formulations in single precision on 1DSfM dataset. With the increased number of LM iterations, a solver is expected to return a more accurate solution, where the accurate is then only limited by the numerical properties of the solver.

Compared to BAL dataset, 1DSfM problems are usually more difficult to solve. In our experiments, our solvers can usually reduce the BAL problems to tolerance $\tau = 0.1$ in 1 or 2 LM iterations, while 5 iterations are required for the 1DSfM problems, therefore it

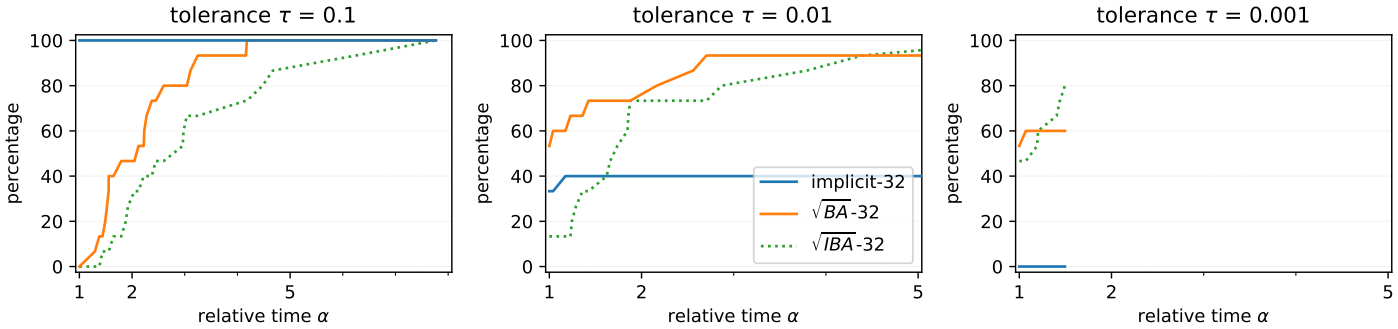


Figure 4.18: Performance profiles of $\sqrt{IBA}$, $\sqrt{BA}$ and implicit SC. It is evaluated on the 1DSfM problems with single precision, and the number of LM iterations is increased to 100.

is a good candidate for testing the accuracy of a solver.

With high tolerance $\tau = 0.1$, the implicit SC solver outperforms both square root solvers with its faster multiplication. However, at tolerance $\tau = 0.01$, implicit SC reaches the accuracy for only 4 problems, while the square root solvers both achieve a much higher accuracy. Moreover, at the lower tolerance $\tau = 0.001$, implicit SC can not reach the accuracy on any of the problems. This result emphasize the superior numerical properties of the square root formulation.

4.8 Memory access pattern

In the last section, we discuss the impact of the better memory access pattern on performance. Although $\sqrt{IBA}$ requires more floating operations than $\sqrt{BA}$, it outperforms $\sqrt{BA}$ with better cache spatial and temporal locality. In this section, we provide more evidence to demonstrate the importance of good memory access pattern.

Compared to $\sqrt{BA}$, $\sqrt{IBA}$ marginalizes landmark variables on a dedicated chunk of memory. In Figure 4.19, we demonstrate the performance difference between pre-allocating the memory once before multiplications, and re-allocating the memory for each landmark per multiplication. In our result, re-allocating the memory is on average 10% slower across all tolerances. The reason of the slower performance could be twofold. As mentioned previously, since marginalization is always performed on the same chuck of memory, it leads to a better temporal locality. Moreover, allocating and deallocating memory usually consume noticeable amount of runtime, therefore real-time programs usually pre-allocate the required memory upfront.

In Section 4.7, we demonstrate the runtime different between $\sqrt{IBA}$ and $\sqrt{BA}$.

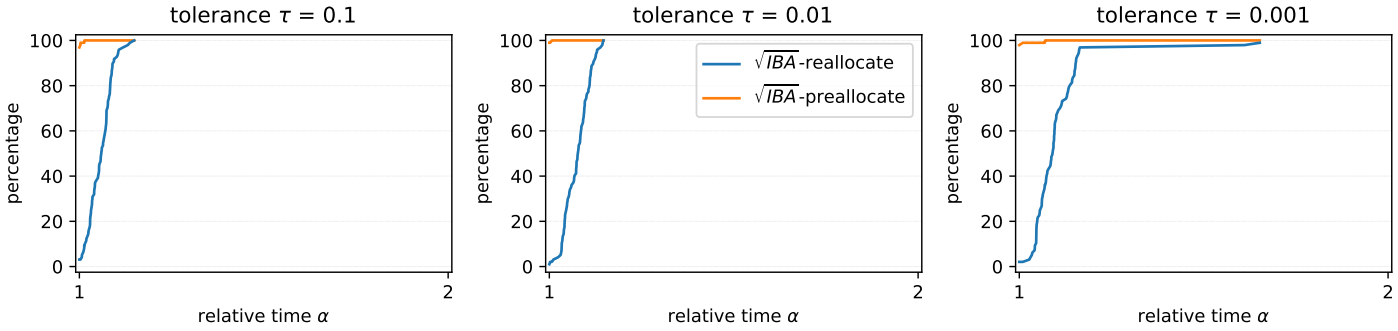


Figure 4.19: Performance profiles of $\sqrt{IBA}$ with pre-allocated and re-allocated memory for marginalizing, evaluated on BAL dataset with single precision. Note that pre-allocating version is used as default in other sections.

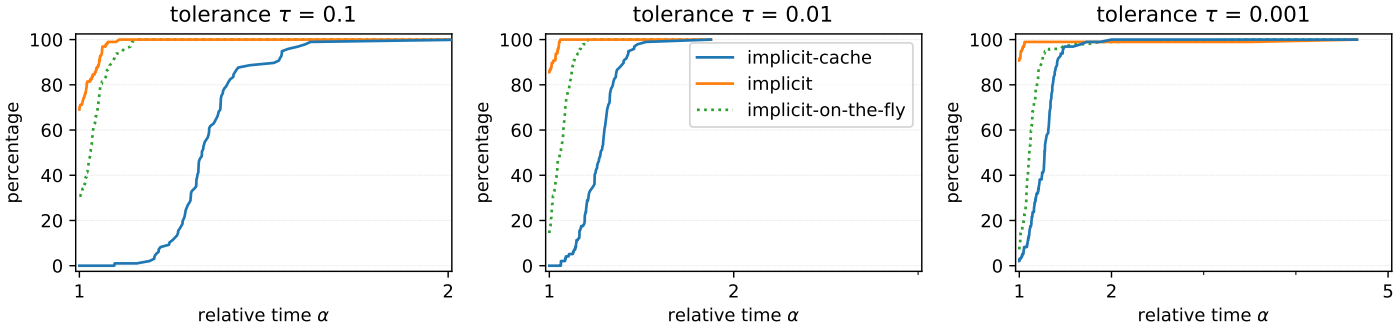


Figure 4.20: Performance profiles of implicit SC solver with different multiplication orders, evaluated on BAL dataset in double precision.

The main difference between $\sqrt{IBA}$ and $\sqrt{BA}$ is the method they obtain $\mathbf{Q}_2^\top \mathbf{J}_p$ for multiplication. We show that the cost of recomputing $\mathbf{Q}_2^\top \mathbf{J}_p$ on demand is only slightly higher than recalling it from memory. Similarly, we study the cost of recalling matrices from memory with implicit SC solver in Figure 4.20. *implicit-cache* constructs $J_p^\top J_l$, then the matrix is recalled from memory for vector-matrix multiplication in (3.1). In contrast, *implicit-on-the-fly* computes $J_p^\top J_l$ for each multiplication.

The implementation of **prepare** and **multiplication** between $\sqrt{IBA}$ and $\sqrt{BA}$ are largely different. In contrast, the only difference between the solvers in Figure 4.20 is the multiplication order, this can better illustrate the cost of recalling matrix from memory.

In the result, *implicit* outperforms the others, where it does not form $J_p^\top J_l$. The

reason is because the time complexity of vector-matrix multiplication is better than matrix-matrix multiplication. Given a square matrix of size $n \times n$, the complexity of vector-matrix multiplication is $O(n^2)$ and matrix-matrix multiplication $O(n^3)$. Despite the cost of reallocating memory for $J_p^\top J_l$, *on-the-fly* variant noticeably outperforms *cache* variant. This experiment indicates that retrieving computed data from memory might be more costly than computing it. This is something to consider while implementing a BA solver, which is also suggested in [Wu+11].

4.9 Power BA

As previously demonstrated, the power BA solver is substantially faster than the other solvers. In this section, we compare power BA to another competitive solver, implicit SC solver.

4.9.1 Performance Analysis

The upper plot in Figure 4.21 shows the time spent in each inner iteration, and the lower plot shows the residual reduction of each inner iteration within one LM iteration. We disable the termination criteria in this experiment and set both the number of LM and inner iterations to 20 for a more accurate one-to-one comparison.

As illustrated in the upper plot, the average duration of inner iterations is relatively similar for both solvers. This implies that if the number of inner iterations is identical, the total runtime spent in the **multiplication** phase for both solvers should be similar. We perform matrix-vector multiplication twice per 10 iterations in our PCG implementation to increase convergence rate. As a result, the runtime of implicit SC solver occasionally spikes. Nonetheless, it should have a negligible impact on the overall runtime of a solver.

The lower plot depicts the residual reduction achieved during each iteration of the LM, where the current residual (2.22) is evaluated at every inner iteration. Ideally, if a solver consistently reduces the residual at each inner iteration, the curve will decline monotonically from 1 to 0 for every 20 inner iterations.

As illustrated in the plot, power BA constantly decreases the residual value during inner iterations. Both solvers achieve a greater reduction in residuals in the first few inner iterations of early LM iterations, as evidenced by the sheer drop following the first inner iteration of each LM iteration. Compared to power BA, the curve of implicit SC sometimes oscillates at some inner iterations. It means if implicit SC stops a few iterations earlier, the residual decrease could be dramatically different. This implies power BA is far more stable for reducing the residual compared to implicit SC. This phenomenon is also often observe in other problems, and it can be more dramatically

than this plot shown. Consequently, the solution computed by implicit SC after a LM iteration may sometimes be higher than initial residual, and the update will be rejected by LM algorithm. In our experiments, the solution computed by power BA is considerably less likely to be rejected by LM than implicit SC.

We demonstrate the runtime of individual phases of power BA and implicit SC solvers in Figure 4.22. Due to the fixed order of power BA, it usually has less inner iterations than the other solvers. Yet, as illustrated in the previous figure, the two solvers should have a similar runtime for **multiplication** when the number of inner iterations is identical.

On the other hand, power BA has significant runtime benefits in the result because it does not need to compute the preconditioner. For power BA, the runtime spent in **prepare** is significantly less than the sum of the **prepare** and **preconditioner** phases for the implicit SC solver, despite the fact that the **prepare** phase of power BA is slightly more expensive due to the additional computation for the diagonal blocks of $\mathbf{H}_p p^{-1}$.

The performance profiles of the power BA and implicit SC solvers are shown in Figure 4.23. The LM and inner iterations are again both set to 20, thus the total number of inner iterations for both solvers is the same. At higher tolerances $\tau = \{0.1, 0.01\}$, power BA outperforms implicit SC, as it is less likely to be rejected by LM and benefits from the absence of a preconditioner. However, given the same number of inner iterations and a low tolerance $\tau = 0.001$, it cannot reach the same level of accuracy as implicit SC. This may indicate that the approximation computed by power BA is less accurate than implicit SC.

In summary, power BA can converge significantly faster than implicit SC to a less accurate solution during the early LM iterations. However, it is less accurate than implicit SC since it cannot achieve the same level of accuracy on most BAL problems as demonstrated in our experiments.

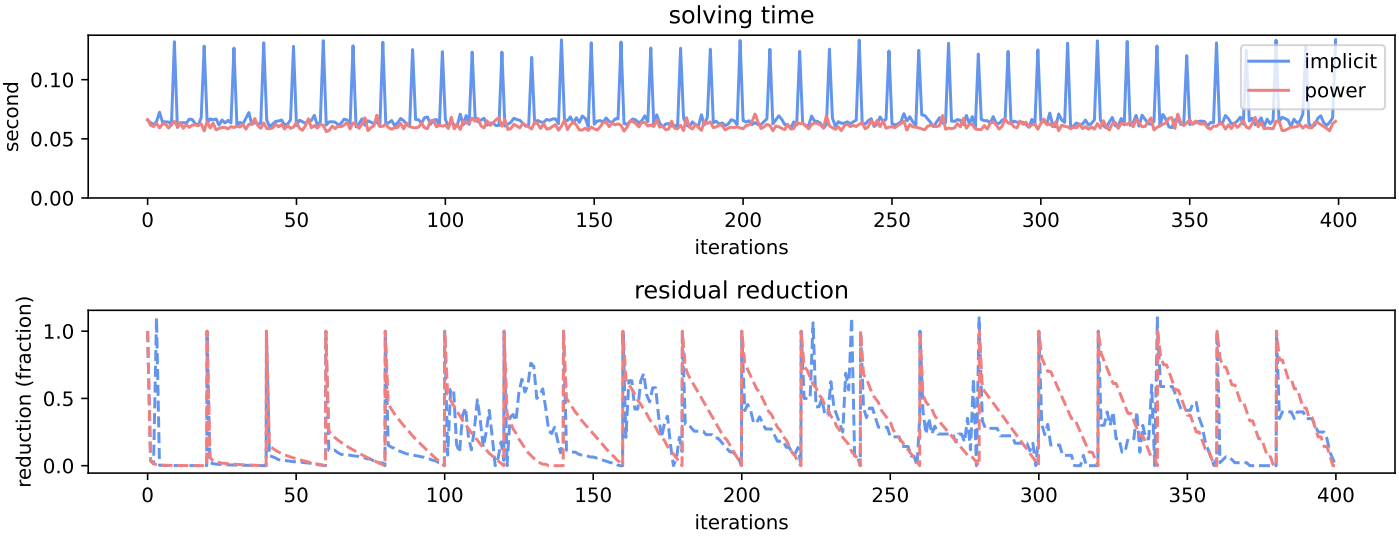
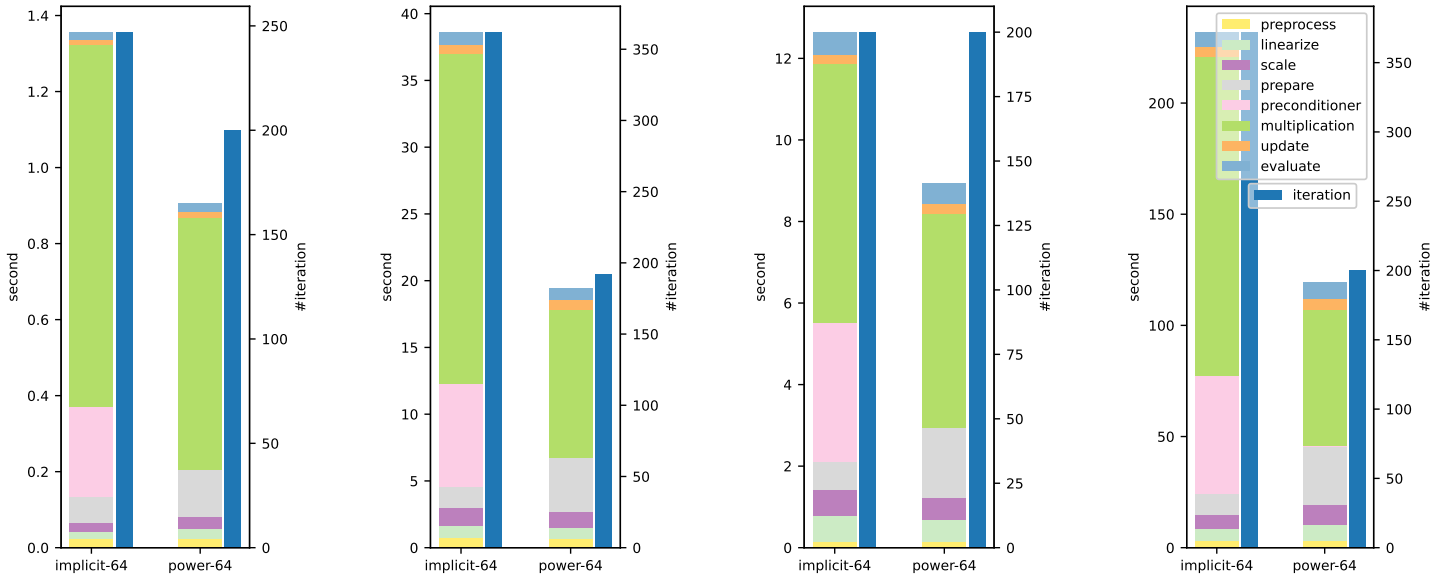


Figure 4.21: Visualization of the inner iterations runtime (a) and reprojection residual reduction (b) of power BA and implicit SC solvers evaluated in double precision on venice1544. The number of LM iterations and inner iterations are both set to 20, hence 400 inner iterations in total. Figure (a) shows the runtime per inner iteration. Figure (b) illustrates the reprojection residual (2.22) reduction per LM iteration. For each inner iterations, we show the percentage of reduction between the error at the start of LM iteration, and the minimal achieved residual across the 20 inner iterations of that LM iteration. 100% indicates no reduction at the current inner iteration, and 0% indicates the error is reduced to the minimal across 20 iterations. The percentage is capped at 1.1 since the error can be larger than the initial error value.



(a) trafalgar50 (small-scale)

(b) venice1184 (medium-scale)

(c) final961 (dense)

(d) final13682 (sparse, large-scale)

Figure 4.22: Total runtime of the individual phases of power BA and implicit SC solvers on four selected BAL problems with different characteristics evaluated in double precision.

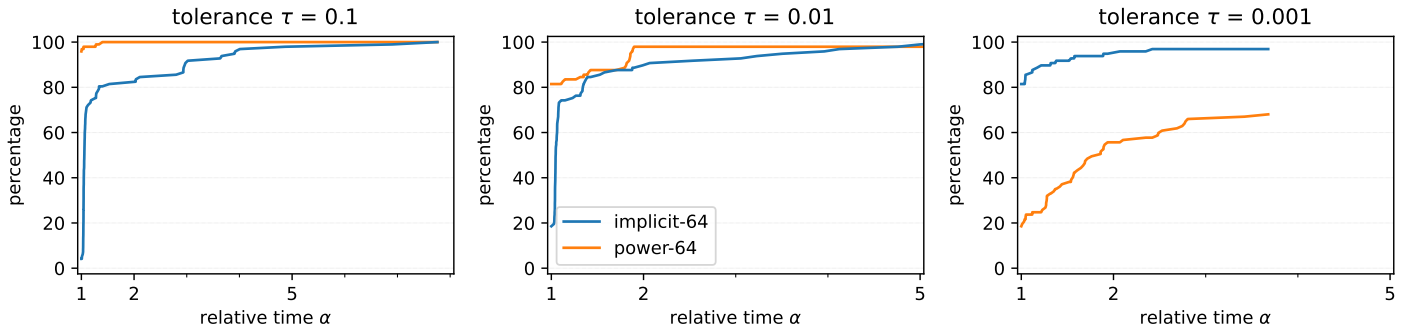


Figure 4.23: Performance profiles of power BA and implicit SC. It is evaluated on the BAL dataset in double precision, and the number of LM and inner iterations are fixed to 20.

5 Conclusion

In this work, we revisit the idea of using factor grouping to accelerate solving BA problems. We propose a new solver by combining factor grouping with the square root formulation. In addition, we present two new solvers: a solver which uses the implicit representation of the square root formulation, and a solver which approximates the solution of the linearized BA problem by power series.

We evaluate and compare the performance of seven different solvers. For fair comparison, we develop an efficient implementation of all solvers that is parallelized and supplemented by SIMD vectorization. The implementation of our solvers is extensively evaluated on three datasets, which cover small- to large-scale and sparse to dense problems.

We study the performance details of the solvers from different perspectives through multiple visualizations. Our research demonstrates that the time complexity of the operations of a solver has a significant impact on performance. Our examination of explicit, implicit, and factor SC solvers reveals that different the time complexity can result in a significantly different runtime.

Based on experiments with multiple settings, we reveal that the memory access pattern of a solver is another influential factor to the performance, which is usually opaque in the result and difficult to verify. As demonstrated in our results, although our implicit square root solver $\sqrt{IBA}$ involves many more floating point operations than $\sqrt{BA}$, it is typically faster or has similar overall performance with $\sqrt{BA}$. The cost of the extra floating point operations in $\sqrt{IBA}$ is offset by the better memory access pattern.

The characteristic of the factor grouping system is thoroughly investigated. In particular, we study the grouping result of the FP-tree as well as the time complexity of explicit and implicit RCS representation. We reveal the influence of the grouping result to the performance of factor grouping scheme in terms of time complexity. While authors of factor grouping focus solely on the time difficulty of PCG multiplication, we find that the time complexity of preparing the linear system has an equivalent impact on performance. We demonstrate that although factor SC is very competitive on a variety of problems, we find that it is often slightly slower than explicit SC on small problems and implicit SC on large ones.

After analyzing the result produced by FP-tree, we notice that factor group scheme

is less suitable for square root formulation as compared to Schur Complement trick. Because landmarks with more observations are treated in the same way as $\sqrt{IBA}$ in $\sqrt{FBA}$, the disadvantage of quadratic complexity of marginalization by QR decomposition becomes more apparent.

In summary, we observe that for large-scale problems, the implicit SC solver is still the de facto choice. For small- to medium-scale problems, we recommend factor SC solver. Although it is slightly slower than explicit SC on smaller problems, it has consistently competitive performance on dense problems, where explicit SC is extremely slow.

For solving a problem in single precision, we recommend the implicit square root solver $\sqrt{IBA}$. In our results, $\sqrt{IBA}$ is generally faster than $\sqrt{BA}$ and $\sqrt{FBA}$. Square root solvers have superior numerical properties to SC solvers, which enables them to compute a more precise solution in single precision at similar speed.

Power BA is recommended for those that can perform optimization in double precision and value speed over accuracy. It is significantly faster than other solvers but has lower accuracy.

Finally, we extensively analyze the influence of time complexity and memory access pattern on performance. They have a significant impact on the performance as demonstrated by our experiments. When designing a BA solver, caution should be exercised to these two aspects.

Bibliography

[Aga+10] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski. "Bundle adjustment in the large." In: *European conference on computer vision*. Springer. 2010, pp. 29–42.

[AMT22] S. Agarwal, K. Mierle, and T. C. S. Team. *Ceres Solver*. Version 2.1. Mar. 2022.

[Car+14] L. Carlone, P. Fernandez Alcantarilla, H.-P. Chiu, Z. Kira, and F. Dellaert. "Mining Structure Fragments for Smart Bundle Adjustment." In: *Proceedings of the British Machine Vision Conference*. BMVA Press, 2014.

[Dem+21] N. Demmel, C. Sommer, D. Cremers, and V. Usenko. "Square Root Bundle Adjustment for Large-Scale Reconstruction." In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021.

[DM02] E. D. Dolan and J. J. Moré. "Benchmarking optimization software with performance profiles." In: *Mathematical programming* 91.2 (2002), pp. 201–213.

[Ead13] E. Eade. "Lie groups for 2d and 3d transformations." In: *URL http://ethaneade.com/lie.pdf, revised Dec* 117 (2013), p. 118.

[G+10] G. Guennebaud, B. Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[Gri+11] G. Grisetti, R. Kümmerle, H. Strasdat, and K. Konolige. "g2o: A general framework for (hyper) graph optimization." In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2011, pp. 9–13.

[Han+07] J. Han, H. Cheng, D. Xin, and X. Yan. "Frequent pattern mining: current status and future directions." In: *Data mining and knowledge discovery* 15.1 (2007), pp. 55–86.

[HPY00] J. Han, J. Pei, and Y. Yin. "Mining frequent patterns without candidate generation." In: *ACM sigmod record* 29.2 (2000), pp. 1–12.

[LA09] M. I. Lourakis and A. A. Argyros. "SBA: A software package for generic sparse bundle adjustment." In: *ACM Transactions on Mathematical Software (TOMS)* 36.1 (2009), pp. 1–30.

[Nas00] S. G. Nash. "A survey of truncated-Newton methods." In: *Journal of computational and applied mathematics* 124.1-2 (2000), pp. 45–59.

[NS90] S. G. Nash and A. Sofer. "Assessing a search direction within a truncated-Newton method." In: *Operations Research Letters* 9.4 (1990), pp. 219–221.

[Phe08] C. Pheatt. "Intel® threading building blocks." In: *Journal of Computing Sciences in Colleges* 23.4 (2008), pp. 298–298.

[Ran04] A. Ranganathan. "The levenberg-marquardt algorithm." In: *Tutoral on LM algorithm* 11.1 (2004), pp. 101–110.

[Ren+21] J. Ren, W. Liang, R. Yan, L. Mai, S. Liu, and X. Liu. *MegBA: A High-Performance and Distributed Library for Large-Scale Bundle Adjustment*. 2021. arXiv: `2112.01349 [cs.CV]`.

[She+94] J. R. Shewchuk et al. *An introduction to the conjugate gradient method without the agonizing pain*. 1994.

[SSS06] N. Snavely, S. M. Seitz, and R. Szeliski. "Photo tourism: exploring photo collections in 3D." In: *ACM SIGGRAPH*. 2006, pp. 835–846.

[Swe] C. Sweeney. *Theia Multiview Geometry Library: Tutorial & Reference*. `http://theia-sfm.org`.

[Tri+99] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzgibbon. "Bundle adjustment—a modern synthesis." In: *International workshop on vision algorithms*. Springer. 1999, pp. 298–372.

[Web+22] S. Weber, N. Demmel, T. C. Chan, and D. Cremers. *Power Bundle Adjustment for Large-Scale 3D Reconstruction*. Manuscript submitted for publication. 2022.

[WS14] K. Wilson and N. Snavely. "Robust Global Translations with 1DSfM." In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2014.

[Wu+11] C. Wu, S. Agarwal, B. Curless, and S. M. Seitz. "Multicore bundle adjustment." In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2011.

[ZXS21] Q. Zheng, Y. Xi, and Y. Saad. "A power Schur complement Low-Rank correction preconditioner for general sparse linear systems." In: *SIAM Journal on Matrix Analysis and Applications* 42.2 (2021), pp. 659–682.