The 2010 IEEE/RSJ International Conference on
Intelligent Robots and Systems
October 18-22, 2010, Taipei, Taiwan

Efficient Sparse Pose Adjustment for 2D Mapping

Kurt Konolige
Willow Garage
Menlo Park, CA 94025
Email: konolige@willowgarage.com

Giorgio Grisetti, Rainer Kümmerle,
Wolfram Burgard
University of Freiburg
Freiburg, Germany
Email: grisetti@informatik.uni-freiburg.de

Benson Limketkai, Regis Vincent
SRI International
Menlo Park, CA 94025
Email: regis.vincent@sri.com

***Abstract*— Pose graphs have become a popular representation for solving the simultaneous localization and mapping (SLAM) problem. A pose graph is a set of robot poses connected by nonlinear constraints obtained from observations of features common to nearby poses. Optimizing large pose graphs has been a bottleneck for mobile robots, since the computation time of direct nonlinear optimization can grow cubically with the size of the graph. In this paper, we propose an efficient method for constructing and solving the linear subproblem, which is the bottleneck of these direct methods. We compare our method, called Sparse Pose Adjustment (SPA), with competing indirect methods, and show that it outperforms them in terms of convergence speed and accuracy. We demonstrate its effectiveness on a large set of indoor real-world maps, and a very large simulated dataset. Open-source implementations in C++, and the datasets, are publicly available.**

I. Introduction

The recent literature in robot mapping shows an increasing interest in graph-based SLAM approaches. In the most general form, the graph has nodes that represent both robot poses and world features, with measurements connecting them as constraints. The goal of all approaches is to jointly optimize the poses of the nodes so as to minimize the error introduced by the constraint. One classical variant of this problem comes from computer vision and is denoted as Bundle Adjustment [25], which is typically solved with a specialized variant of the Levenberg-Marquardt (LM) nonlinear optimizer. In the SLAM literature, Lu-Milios [18], GraphSLAM [24], and $\sqrt{\text{SAM}}$ [4] are all variants of this technique.

Since features tend to outnumber robot poses, more compact systems can be created by converting observations of features into direct constraints among the robot poses, either by marginalization [1, 24, 4], or by direct matching – for example, matching laser scans between two robot poses yields a relative pose estimate for the two. Pose constraint systems, in typical robotic mapping applications, exhibit a sparse structure of connections, since the range of the sensor is typically limited to the vicinity of the robot.

Solving pose graphs efficiently (i.e., finding the optimal positions of the nodes) is a key problem for these methods especially in the context of online mapping problems. A typical 2D laser map for a 100m x 100m office space may have several thousand nodes and many more constraints (see Figure 1). Furthermore, adding a loop closure constraint to this map can affect almost all of the poses in the system.

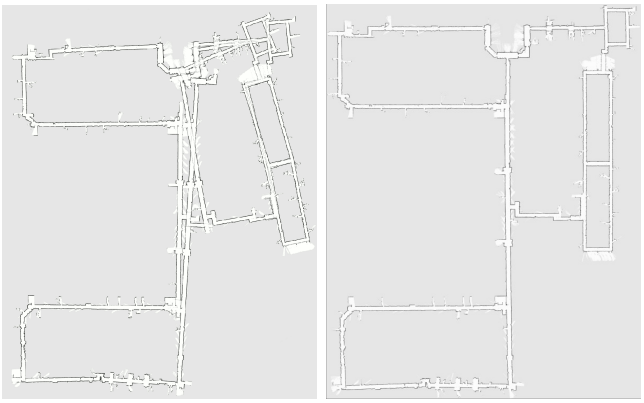

Fig. 1. Large MIT corridor map, unoptimized (left) and optimized (right). A full nonlinear optimization of this map (3603 nodes and 4986 constraints), starting from the odometry positions of the left-side figure, takes just 150 ms with our method.

At the heart of the LM method lies the solution of a large sparse linear problem. In this paper, we develop a method to efficiently compute the sparse matrix from the constraint graph, and use direct sparse linear methods to solve it. In analogy to Sparse Bundle Adjustment in the vision literature, we call this method Sparse Pose Adjustment (SPA), since it deals with the restricted case of pose-pose constraints. The combination of an SBA/GraphSLAM optimizer with efficient methods for solving the linear subproblem has the following advantages.

- It takes the covariance information in the constraints into account which leads to more accurate solutions.
- SPA is robust and tolerant to initialization, with very low failure rates (getting stuck in local minima) for both incremental and batch processing.
- Convergence is very fast as it requires only a few iterations of the LM method.
- Unlike EKF and information filters, SPA is fully nonlinear: at every iteration, it linearizes all constraints around their current pose.
- SPA is efficient in both batch and incremental mode.

We document these and other features of the method in the experimental results section where we also compare our method to other LM and non-LM state-of-the-art optimizers.

One of the benefits of the efficiency of SPA is that a mapping system can continuously optimize its graph, providing the best global estimate of all nodes, with very little

computational overhead. Solving the optimization problem for the large map shown in Figure 1 requires only 150 ms from an initial configuration provided by odometry. In the incremental mode, where the graph is optimized after each node is added, it requires less than 15 ms for any node addition.

Although SPA can be parameterized with 3D poses, for this paper we have restricted it to 2D mapping, which is a well-developed field with several competing optimization techniques. Our intent is to show that a 2D pose-based mapping system can operate on-line using SPA as its optimization engine, even in large-scale environments and with large loop closures, without resorting to submaps or complicated partitioning schemes.

II. Related Work

Lu and Milios [18] presented the seminal work on graph-based SLAM, where they determine the pairwise measurements between scans via ICP scan-matching and then optimize the graph by iterative linearization. At that time, efficient optimization algorithms were not available to the SLAM community and graph-based approaches were regarded as too time-consuming. Despite this, the intuitive formulation of graph-based SLAM attracted many researchers with valuable contributions. Gutmann and Konolige [12] proposed an effective way for constructing such a network and for detecting loop closures while running an incremental estimation algorithm.

Since the Lu and Milios paper, many approaches for graph optimization have been proposed. Howard *et al.* [13] apply relaxation to localize the robot and build a map. Duckett *et al.* [6] propose the usage of Gauss-Seidel relaxation to minimize the error in the network of constraints. To overcome the inherently slow convergence of relaxation methods, Frese *et al.* [9] propose a variant of Gauss-Seidel relaxation called multi-level relaxation (MLR). It applies relaxation at different resolutions. MLR is reported to provide very good results in 2D environments, especially if the error in the initial guess is limited.

Olson *et al.* [21] proposed stochastic gradient descent to optimize pose graphs. This approach has the advantage of being easy to implement and exceptionally robust to wrong initial guesses. Later, Grisetti *et al.* [10] extended this approach by applying a tree based parameterization that significantly increases the convergence speed. The main problem of these approaches is that they assume the error in the graph to be more or less uniform, and thus they are difficult to apply to graphs where some constraints are under-specified.

The most intuitive way to optimize a graph is probably by nonlinear least-squares optimization, such as LM. Least-squares methods require to repetitively solve a large linear system obtained by linearizing the original likelihood function of the graph. This linear system is usually very large; accordingly, the first graph-based approaches were time consuming, because they did not exploit its natural sparsity. One promising technique is Preconditioned Conjugate Gradient (PCG) [2], which was later used by Konolige [15] and Montemerlo and Thrun [20] as an efficient solver for large sparse pose constraint systems; the preconditioner is typically incomplete Cholesky decomposition. PCG is an iterative method, which in general requires n iterations to converge, where n is the number of variables in the graph. We have implemented a sparse-matrix version of PCG from Sparselib++ and IML++ [5], and use this implementation for comparison experiments in this paper.

More recently, Dellaert and colleagues use bundle adjustment, which they implement using sparse direct linear solvers [3]; they call their system $\sqrt{\text{SAM}}$ [4]). Our approach is similar to $\sqrt{\text{SAM}}$; we differ from their approach mostly in engineering, by efficient construction of the linear subproblem using ordered data structures. We also use LM instead of a standard nonlinear least-square method, thereby increasing robustness. Finally, we introduce a "continuable LM" method for the incremental case, and an initialization method that is a much more robust approach to the batch problem.

Kaess et al. [14] introduced a variant of $\sqrt{\text{SAM}}$, called iSAM, that performs incremental update of the linear matrix associated with the nonlinear least-squares problem. Relinearization and variable ordering are performed only occasionally, thereby increasing computational efficiency. In our approach, relinearization and matrix construction are very efficient, so such methods become less necessary. Currently we do not have an implementation of either iSAM or $\sqrt{\text{SAM}}$ to test against for performance.

Relaxation or least-squares approaches proceed by iteratively refining an initial guess. Conversely, approaches based on stochastic gradient descent are more robust to the initial guess. In the SLAM literature the importance of this initial guess has been often underestimated. The better the initial guess is, the more likely it is for an algorithm to find the correct solution. In this paper, we address this point and evaluate three different strategies for computing the initial guess.

In contrast to full nonlinear optimization, several researchers have explored filtering techniques to solve the graphs incrementally, using an information matrix form. The first such approach was proposed by Eustice *et al.* and denoted Delayed Sparse Information Filter (DSIF) [7]. This technique can be very efficient, because it adds only a small constant number of elements to the system information matrix, even for loop closures. However, recovering the global pose of all nodes requires solving a large sparse linear system; there are faster ways of getting approximate recent poses.

Frese proposed the TreeMap [8] algorithm that captures the sparse structure of the system by a tree representation. Each leaf in the tree is a local map and the consistency of the estimate is achieved by sending updates to the local maps through the tree. Under ideal conditions, this approach can update the whole map in $O(n \log n)$ time, where n is the numbers of elements in the map. However, if the map has many local connections the size of the local maps can be very large and their updates (which are regarded as elementary operations) become computationally expensive as shown in the remainder of this paper.

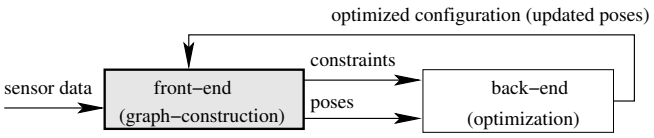

Fig. 2. Typical graph-based SLAM system. The front-end and the back end are executed in an interleaved way.

To summarize the paper: we propose an efficient approach for optimizing 2D pose graphs that uses direct sparse Cholesky decomposition to solve the linear system. The linear system is computed in a memory-efficient way that minimizes cache misses and thus significantly improves the performance. We compare our method, in both accuracy and speed, to existing LM and non-LM approaches that are avaiable, and show that SPA outperforms them. Open source implementations are available both in C++ and in matlab/octave.

Efficient direct (non-iterative) algorithms to solve sparse systems have become available [3], thus reviving a series of approaches for optimizing the graphs which have been discarded in the past. In this paper,

III. System Formulation

Popular approaches to solve the SLAM problem are the so-called "graph-based" or "network-based" methods. The idea is to represent the history of the robot measurements by a graph. Every node of the graph represents a sensor measurement or a local map and it is labeled with the location at which the measurement was taken. An edge between two nodes encodes the spatial information arising from the alignment of the connected measurements and can be regarded as a spatial constraint between the two nodes.

In the context of graph-based SLAM, one typically considers two different problems. The first one is to identify the constraints based on sensor data. This so-called data association problem is typically hard due to potential ambiguities or symmetries in the environment. A solution to this problem is often referred to as the SLAM front-end and it directly deals with the sensor data. The second problem is to correct the poses of the robot to obtain a consistent map of the environment *given* the constraints. This part of the approach is often referred to as the optimizer or the SLAM back-end. Its task is to seek for a configuration of the nodes that maximizes the likelihood of the measurements encoded in the constraints. An alternative view to this problem is given by the spring-mass model in physics. In this view, the nodes are regarded as masses and the constraints as springs connected to the masses. The minimal energy configuration of the springs and masses describes a solution to the mapping problem.

During its operation a graph-based SLAM system interleaves the execution of the front-end and of the back-end, as shown in Figure 2. This is required because the front-end needs to operate on a partially optimized map to restrict the search about potential constraints. The more accurate the current estimate is, the more robust the constraints generated by the front-end will be and the faster its operation. Accordingly, the performance of the optimization algorithm, measured in terms of accuracy of the estimate and execution time, has a major impact on the overall mapping system.

In this paper we describe in detail an efficient and compact optimization approach that operates on 2D graphs. Our algorithm can be coupled with arbitrary front-ends that handle different kinds of sensors. For clarity of presentation we shortly describe a front-end for laser data. However, the general concepts can be straightforwardly applied to different sensors.

IV. Sparse Pose Adjustment

To optimize a set of poses and constraints, we use the well-known Levenberg-Marquardt (LM) method as a framework, with particular implementations that make it efficient for the sparse systems encountered in 2D map building. In analogy to the Sparse Bundle Adjustment of computer vision, which is a similarly efficient implementation of LM for cameras and features, we call our system Sparse Pose Adjustment (SPA).

A. Error Formulation

The variables of the system are the set of global poses $\mathbf{c}$ of the robot, parameterized by a translation and angle: $c_i = [t_i, \theta_i] = [x_i, y_i, \theta_i]^\top$. A *constraint* is a measurement of one node c_j from another's (c_i) position. The measured offset between c_i and c_j, in c_i's frame, is $\bar{z}_{ij}$, with precision matrix Λ_{ij} (inverse of covariance). For any actual poses of c_i and c_j, their offset can be calculated as

$$h(c_i, c_j) \equiv \begin{cases} R_i^\top (t_j - t_i) \\ \theta_j - \theta_i \end{cases} \tag{1}$$

Here R_i is the 2x2 rotation matrix of θ_i. $h(c_i, c_j)$ is called the *measurement equation*.

The error function associated with a constraint, and the total error, are

$$\begin{aligned} e_{ij} &\equiv \bar{z}_{ij} - h(c_i, c_j) \\ \chi^2(\mathbf{c}, \mathbf{p}) &\equiv \sum_{ij} e_{ij}^\top \Lambda_{ij} e_{ij} \end{aligned} \tag{2}$$

Note that the angle parameter in $h(c_i, c_j)$ is not unique, since adding or subtracting 2π yields the same result. Angle differences are always normalized to the interval $(-\pi, \pi]$ when they occur.

B. Linear System

The optimal placement of $\mathbf{c}$ is found by minimizing the total error in Equation 2. A standard method for solving this problem is Levenberg-Marquardt (LM), iterating a linearized solution around the current values of $\mathbf{c}$. The linear system is formed by stacking the variables $\mathbf{c}$ into a vector $\mathbf{x}$, and the error functions into a vector $\mathbf{e}$. Then we define:

$$\begin{aligned} \mathbf{\Lambda} &\equiv \begin{bmatrix} \Lambda_{ab} & & \\ & \ddots & \\ & & \Lambda_{mn} \end{bmatrix} \\ \mathbf{J} &\equiv \frac{\partial \mathbf{e}}{\partial \mathbf{x}} \\ \mathbf{H} &\equiv \mathbf{J}^\top \mathbf{\Lambda} \mathbf{J} \end{aligned} \tag{3}$$

The LM system is:

$$(\mathbf{H} + \lambda\, diag\mathbf{H})\, \Delta\mathbf{x} = \mathbf{J}^\top \mathbf{\Lambda} \mathbf{e} \tag{4}$$

Here λ is a small positive multiplier that transitions between gradient descent and Newton-Euler methods. Gradient descent is more robust and less likely to get stuck in local minima, but converges slowly; Newton-Euler has the opposite behavior.

The matrix $\mathbf{H}$ is formed by adding four components for each measurement $h(c_i, c_j)$:

$$\begin{array}{ccc} \ddots & & \\ J_i^\top \Lambda_{ij} J_i & \cdots & J_i^\top \Lambda_{ij} J_j \\ \vdots & \ddots & \vdots \\ J_j^\top \Lambda_{ij} J_i & \cdots & J_j^\top \Lambda_{ij} J_j \\ & & \ddots \end{array} \tag{5}$$

Here we have abused the notation for J slightly, with J_i being the Jacobian of e_{ij} with respect to the variable c_i. The components are all 3x3 blocks. The right-hand side is formed by adding 3x1 blocks $J_{c_i} \Lambda_{ij} e_{ij}$ and $J_{c_j} \Lambda_{ij} e_{ij}$ for each constraint.

Solving the linear equation yields an increment $\Delta\mathbf{x}$ that can be added back in to the current value of $\mathbf{x}$ as follows:

$$\begin{aligned} t_i &= t_i + \Delta t_i \\ \theta_i &= \theta_i + \Delta\theta_i \end{aligned} \tag{6}$$

C. Error Jacobians

Jacobians of the measurement function h appear in the normal equations (4), and we list them here.

$$\begin{aligned} \frac{\partial e_{ij}}{\partial t_i} &\equiv \begin{bmatrix} -R_i^\top \\ 0\ 0 \end{bmatrix} & \frac{\partial e_{ij}}{\partial \theta_i} &\equiv \begin{bmatrix} -\partial R_i^\top / \partial \theta_i\ (t_j - t_i) \\ -1 \end{bmatrix} \\ \frac{\partial e_{ij}}{\partial t_j} &\equiv \begin{bmatrix} R_i^\top \\ 0\ 0 \end{bmatrix} & \frac{\partial e_{ij}}{\partial \theta_j} &\equiv [0\ 0\ 1]^\top \end{aligned} \tag{7}$$

D. Sparsity

We are interested in large systems, where the number of poses $||\mathbf{c}||$ can be 10k or more (the largest real-world indoor dataset we have been able to find is about 3k poses, but we can generate synthetic datasets of any order). The number of system variables is $3||\mathbf{c}||$, and the $\mathbf{H}$ matrix is $||\mathbf{c}||^2$, or over 10^8 elements. Manipulating such large matrices is expensive. Fortunately, for typical scenarios the number of constraints grows only linearly with the number of poses, so that $\mathbf{H}$ is very sparse. We can take advantage of the sparsity to solve the linear problem more efficiently.

For solving (4) in sparse format, we use the CSparse package [3]. This package has a highly-optimized Cholesky decomposition solver for sparse linear systems. It employs several strategies to decompose $\mathbf{H}$ efficiently, including a logical ordering and an approximate minimal degree (AMD) algorithm to reorder variables when $\mathbf{H}$ is large.

In general the complexity of decomposition will be $O(n^3)$ in the number of variables. For sparse matrices, the complexity will depend on the density of the Cholesky factor, which in turn depends on the structure of $\mathbf{H}$ and the order of its variables. Mahon et al. [19] have analyzed the behavior of the Cholesky factorization as a function of the loop closures in the SLAM system. If the number of loop closures is constant, then the Cholesky factor density is $O(n)$, and decomposition is $O(n)$. If the number of loop closures grows linearly with the number of variables, then the Cholesky factor density grows as $O(n^2)$ and decomposition is $O(n^3)$.

E. Compressed Column Storage

Each iteration of the LM algorithm has three steps: setting up the linear system, decomposing $\mathbf{H}$, and finding Δx by back-substitution. Setting up the system is linear in the number of constraints (and hence in the number of variables for most graph-based SLAM systems). In many situations it can be the more costly part of the linear solver. Here we outline an efficient method for setting up the sparse matrix form of $\mathbf{H}$ from the constraints generated by Equation (5).

CSparse uses *compressed column storage* (CCS) format for sparse matrices. The figure below shows the basic idea.

$$\begin{bmatrix} 1 & 0 & 4 & 0 \\ 0 & 5 & 0 & 2 \\ 0 & 0 & 0 & 1 \\ 6 & 8 & 0 & 0 \end{bmatrix} \Rightarrow$$

col_ptr	0		2		4	5		7
row_ind	0	3	1	3	0	1	2	
val	1	6	5	8	4	2	1	

(8)

Each nonzero entry in the array is placed in the *val* vector. Entries are ordered by column first, and then by row within the column. *col_ptr* has one entry for each column, plus a last entry which is the number of total nonzeros (*nnz*). The *col_ptr* entry for a column points to the start of the column in the *row_ind* and *val* variables. Finally, *row_ind* gives the row index of each entry within a column.

CCS format is storage-efficient, but is difficult to create incrementally, since each new nonzero addition to a column causes a shift in all subsequent entries. The most efficient way would be to create the sparse matrix in column-wise order, which would require cycling through the constraints $||\mathbf{c}||$ times. Instead, we go through the constraints just once, and store each 3x3 block $J_i^\top \Lambda_{ij} J_i$ in a special block-oriented data structure that parallels the CCS format. The algorithm is given in Table I. In this algorithm, we make a pass through the constraints to store the 3x3 block matrices into C++ std::map data structures, one for each column. Maps are efficient at ordered insertion based on their keys, which is the row index. Once this data structure is created (step (2)), we use the ordered nature of the maps to create the sparse CCS format of $\mathbf{H}$ by looping over each map in the order of its keys, first to create the column and row indices, and then to put in the values. The reason for separating the column/row creation from value insertion is because the former only has to be done once for any set of iterations of LM.

Note that only the upper triangular elements of $\mathbf{H}$ are stored, since the Cholesky solver in CSparse only looks at this part, and assumes the matrix is symmetric.

TABLE I

ALGORITHM FOR SETTING UP THE SPARSE $\mathbf{H}$ MATRIX

$\mathbf{H} = \textbf{CreateSparse}(\mathbf{e}, \mathbf{c}_f)$

Input: set of constraints e_{ij}, and a list of free nodes (variables)
Output: sparse upper triangular $\mathbf{H}$ matrix in CCS format

1) Initialize a vector of size $||\mathbf{c}_f||$ of C++ std::map's; each map is associated with the corresponding column of $\mathbf{H}$. The key of the map is the row index, and the data is an empty 3x3 matrix. Let map$[i, j]$ stand for the j'th entry of the i'th map.
2) For each constraint e_{ij}, assuming $i < j$:
   a) In the steps below, create the map entries if they do not exist.
   b) If c_i is free, map$[i, i]$ += $J_i^\top \Lambda_{ij} J_i$.
   c) If c_j is free, map$[j, j]$ += $J_j^\top \Lambda_{ij} J_j$.
   d) If c_i, c_j are free, map$[j, i]$ += $J_i^\top \Lambda_{ij} J_j$.
3) Set up the sparse upper triangular matrix $\mathbf{H}$.
   a) In the steps below, ignore elements of the 3x3 map$[i, i]$ entries that are below the diagonal.
   b) Go through map[] in column then row order, and set up *col_ptr* and *row_ind* by determining the number of elements in each column, and their row positions.
   c) Go through map[] again in column then row order, and insert entries into *val* sequentially.

F. Continuable LM System

The LM system algorithm is detailed in Table II. It does one step in the LM algorithm, for a set of nodes $\mathbf{c}$ with associated measurements. Running a single iteration allows for incremental operation of LM, so that more nodes can be added between iterations. The algorithm is *continuable* in that λ is saved between iterations, so that successive iterations can change λ based on their results. The idea is that adding a few nodes and measurements doesn't change the system that much, so the value of λ has information about the state of gradient descent vs. Euler-Newton methods. When a loop closure occurs, the system can have trouble finding a good minima, and λ will tend to rise over the next few iterations to start the system down a good path.

There are many different ways of adjusting λ; we choose a simple one. The system starts with a small lambda, 10^{-4}. If the updated system has a lower error than the original, λ is halved. If the error is the same or larger, λ is doubled. This works quite well in the case of incremental optimization. As long as the error decreases when adding nodes, λ decreases and the system stays in the Newton-Euler region. When a link is added that causes a large distortion that does not get corrected, λ can rise and the system goes back to the more robust gradient descent.

V. SCAN MATCHING

SPA requires precision (inverse covariance) estimates from matching of laser scans (or other sensors). Several scan-match algorithms can provide this, for example, Gutmann *et al.* [11] use point matches to lines extracted in the reference scan, and return a Gaussian estimate of error. More recently, the correlation method of Konolige and Chou [17], extended by Olson [22], provides an efficient method for finding the globally best match within a given range, while returning

TABLE II

CONTINUABLE LM ALGORITHM

ContinuableLM$(\mathbf{c}, \mathbf{e}, \lambda)$

Input: nodes $\mathbf{c}$ and constraints $\mathbf{e}$, and diagonal increment λ
Output: updated $\mathbf{c}$

1) If $\lambda = 0$, set λ to the stored value from the previous run.
2) Set up the sparse $\mathbf{H}$ matrix using CreateSparse$(\mathbf{e}, \mathbf{c} - c_0)$, with c_0 as the fixed pose.
3) Solve $(\mathbf{H} + \lambda\, diag\mathbf{H})\, \Delta\mathbf{x} = \mathbf{J}^\top \mathbf{\Lambda} \mathbf{e}$, using sparse Cholesky with AMD.
4) Update the variables $\mathbf{c} - c_0$ using Equation (6).
5) If the error $\mathbf{e}$ has decreased, divide λ by two and save, and return the updated poses for $\mathbf{c} - c_0$.
6) If the error $\mathbf{e}$ has increased, multiply λ by two and save, and return the original poses for $\mathbf{c} - c_0$.

an accurate covariance. The method allows either a single scan or set of aligned scans to be matched against another single scan or set of aligned scans. This method is used in the SRI's mapping system Karto[1] for both local matching of sequential scans, and loop-closure matching of sets of scans as in [12]. To generate the real-world datasets for experiments, we ran Karto on 63 stored robot logs of various sizes, using its scan-matching and optimizer to build a map and generate constraints, including loop closures. The graphs were saved and used as input to all methods in the experiments.

VI. EXPERIMENTS

In this section, we present experiments where we compare SPA with state-of-the art approaches on 63 real world datasets and on a large simulated dataset. We considered a broad variety of approaches, including the best state-of-the-art.

- Information filter: DSIF [7]
- Stochastic gradient descent: TORO [10]
- Decomposed nonlinear system: Treemap [8]
- Sparse pose adjustment: SPA, with (a) sparse direct Cholesky solver, and (b) iterative PCG [15]

We updated the PCG implementation to use the same "continued LM" method as SPA; the only difference is in the underlying linear solver. The preconditioner is an incomplete Cholesky method, and the conjugate gradient is implemented in sparse matrix format.

We also evaluated a dense Cholesky solver, but both the computational and the memory requirements were several orders of magnitude larger than the other approaches. As an example, for a dataset with 1600 constraints and 800 nodes one iteration using a dense Cholesky solver would take 2.1 seconds while the other approaches require an average of a few milliseconds. All experiments are executed on an Intel Core i7-920 running at 2.67 Ghz.

In the following, we report the cumulative analysis of the behavior of the approaches under different operating conditions; results for all datasets are available online at `www.ros.org/research/2010/spa`. We tested each method both in batch mode and on-line. In batch mode, we

[1]Information on Karto can be found at www.kartorobotics.com.

Fig. 3. Effect of the χ^2 reduction. This figure shows two maps generated from two graphs having a different χ^2 error. The error of the graph associated to the top map is 10 times bigger than the one on the bottom. Whereas the overall structure appears consistent in both cases, the details in the map with the lower χ^2 appear significantly more consistent.

provided the algorithm with the full graph while in on-line mode we carried out a certain number of iterations whenever a new node was added to the graph. In the remainder of this section we first discuss the off-line experiments, then we present the on-line experiments. We conclude by analyzing all methods on a large-scale simulated dataset.

A. Accuracy Measure

For these indoor datasets, there is no ground truth. Instead, the measure of goodness for a pose-constraint system is the covariance-weighted squared error of the constraints, or χ^2 error. If the scan matcher is accurate, lower χ^2 means that scans align better. Figure 3 shows this effect on a real-world dataset.

B. Real-World Experiments: Off-Line Optimization

To optimize a dataset off-line, we provide each optimizer with a full description of the problem. We leave out from the comparison DSIF and TreeMap, since they only operate incrementally (DSIF is equivalent to one iteration of SPA in batch mode). Since the success of the off-line optimization strongly depends on the initial guess, we also investigated two initialization strategies, described below.

- **Odometry**: the nodes in the graph are initialized with incremental constraints. This is a standard approach taken in almost all graph optimization algorithms.
- **Spanning-Tree**: A spanning tree is constructed on the graph using a breadth-first visit. The root of the tree is the first node in the graph. The positions of the nodes are initialized according to a depth-first visit of the spanning tree. The position of a child is set to the position of the parent transformed according to the connecting constraint. In our experiments, this approach gives the best results.

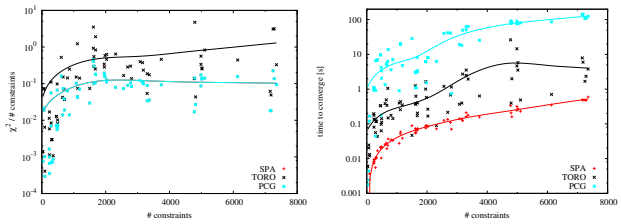

Fig. 4. Batch optimization on real-world datasets, using Odometry and Spanning-Tree initialization. Left: the final χ^2 error per constraint for all the approaches. Right: the time required to converge. Every point in the plots represents one dataset. The datasets are sorted according to the number of constraints in the graph, which effectively measures the complexity of the optimization. The curves show the interpolated behavior of each method, for better visualization. Note that PCG and SPA converge to the same error.

For each dataset and each optimizer we computed the initial guesses described above. Every optimizer was run for a minimum number of iterations or until a termination criterion was met. We measured the time required to converge and the χ^2 error for each approach. The results are summarized in Figure 4 for the Odometry and Spanning-Tree initializations. For these datasets, there was no substantial difference in performance between the two types of initialization.

In the error graph, PCG and SPA converged to almost exactly the same solution, since the only difference is the linear solver. They both dominate TORO, which has more than 10 times the error for the larger graphs. We attribute this to the inability of TORO to handle non-spherical covariances, and its very slow convergence properties. SPA requires almost an order of magnitude less computational effort than PCG or TORO for almost all graphs.

TORO was designed to be robust to bad initializations, and to test this we also ran all methods with all nodes initialized to (0,0,0). In this case, SPA and PCG converged to non-global minima for all datasets, while TORO was able to reconstruct the correct topology.

C. Real-World Experiments: On-Line Optimization

For the on-line comparison, we incrementally augment the graph by adding one node and by connecting the newly added node to the previously existing graph. We invoke the optimizer after inserting each node, and in this way simulate its behavior when executed in conjunction with a SLAM front-end. The optimization is carried out for a maximum number iterations, or until the error does not decrease. The maximum number of iterations for SPA/PCG is 1; for TreeMap, 3; and for TORO, 100. Since PCG iteratively solves the linear subproblem, we limited it to 50 iterations there. These thresholds were selected to obtain the best performances in terms of error reduction. In Figure 5 we report the statistics on the execution time and on the error per constraint every time a new constraint was added.

In terms of convergence, SPA/PCG dominate the other methods. This is not surprising in the case of DSIF, which is an information filter and therefore will be subject to linearization errors when closing large loops. TORO has the closest performance to SPA, but suffers from very slow convergence per iteration, characteristic of gradient methods; it also does

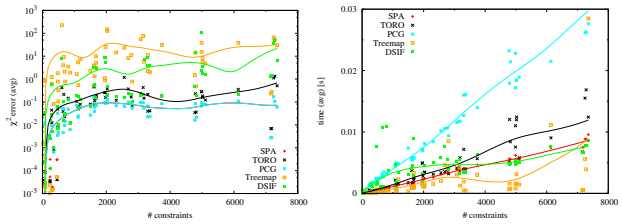

Fig. 5. On-line optimization on real-world datasets. Left: the average χ^2 error per constraint after adding a node during incremental operation. Right: the average time required to optimize the graph after inserting a node. Each data point represents one data set, the x-axis shows the total number of constraints of that data set. Note that the error for PCG and SPA is the same in the left graph.

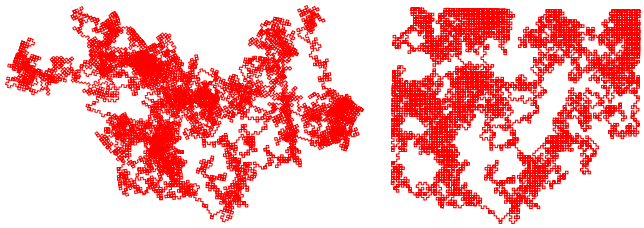

Fig. 6. Large simulated dataset containing 100k nodes and 400k constraints used in our experiments. Left: initial guess computed from the odometry. Right: optimized graph. Our approach requires about 10 seconds to perform a full optimization of the graph when using the spanning-tree as initial guess.

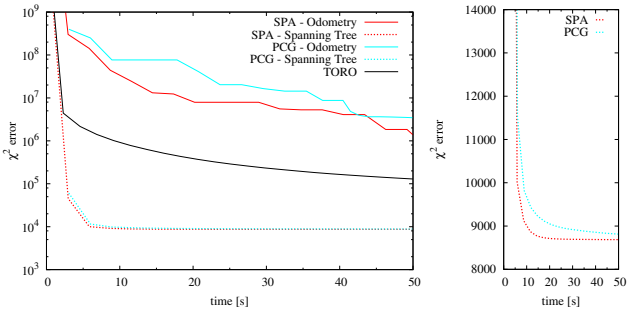

Fig. 7. Evolution of the χ^2 error during batch optimization of a large simulated dataset consisting of 100,000 nodes and 400,000 constraints, under odometry and spanning-tree initialization. The left plot shows the overall execution, while the right plot shows a magnified view of the χ^2 error of SPA and PCG close to the convergence point.

not handle non-circular covariances, which limit its ability to achieve a minimal χ^2. Treemap is much harder to analyze, since it has a complicated strategy setting up a tree structure for optimization. For these datasets, it appears to have a large tree (large dataset loops) with small leaves. The tree structure is optimized by fixing linearizations and deleting connections, which leads to fast computation but poor convergence, with χ^2 almost 3 orders of magnitude worse than SPA.

All the methods are approximately linear overall in the size of the constraint graph, which implies that the number of large loop closures grows slowly. Treemap has the best performance over all datasets, followed by SPA and DSIF. Note that SPA is extremely regular in its behavior: there is little deviation from the linear progression on any dataset. Furthermore that average and max times are the same: see the graph in Figure 8. Finally, TORO and PCG use more time per iteration, with PCG about four times that of SPA. Given SPA's fast convergence, we could achieve even lower computational effort by applying it only every n node additions. We emphasize that these graphs were the largest indoor datasets we could find, and they are not challenging for SPA.

D. Synthetic Dataset

To estimate the asymptotic behavior of the algorithms we generated a large simulated dataset. The robot moves on a grid; each cell of the grid has a side of 5 meters, and we create a node every meter. The perception range of the robot is 1.5 meters. Both the motion of the robot and the measurements are corrupted by a zero mean Gaussian noise with standard deviation $\sigma_u = diag(0.01\,\text{m}, 0.01\,\text{m}, 0.5\,\text{deg})$. Whenever a robot is in the proximity of a position it has visited, we generate a new constraint. The simulated area has spans over 500×500 meters, and the overall trajectory is 100 km long. This results in frequent re-observations. The full graph is shown in Figure 6. This is an extremely challenging dataset, and much worse than any real-world dataset. In the following we report the result of batch and on-line execution of all the approaches we compared.

a) Off-Line Optimization: Each batch approach was executed with the three initializations described in the previous section: odometry, spanning-tree, and zero. Results are shown in Figure 7 as a function of time. The only approach which is able to optimize the graph from a zero or odometry initialization is TORO; SPA/PCG essentially does not converge to a global minimum under odometry or zero starts. SPA/PCG converges globally from the spanning-tree initialization after 10 seconds or so, with SPA being significantly faster at the convergence point (see magnified view in Figure 7). TORO has good initial convergence, but has a long tail because of gradient descent.

b) On-Line Optimization: We processed the dataset incrementally, as described in Section VI-C. In Figure 8 we report the evolution of the χ^2 error and time per added node. Both SPA and TreeMap converge to a minimum χ^2 (see Figure 7 for the converged map). However, their computational behavior is very different: TreeMap can use up to 100 seconds per iteration, while SPA grows slowly with the size of the graph. Because of re-visiting in the dataset, TreeMap has a small tree with very large leaves, and perform LM optimization at each leaf, leading to low error and high computation.

The other methods have computation equivalent to SPA, but do not converge as well. Again DSIF performs poorly, and does not converge. TORO converges but as usual has difficulty with cleaning up small errors. PCG spikes because it does not fully solve the linear subproblem, eventually leading to higher overall error.

VII. Conclusions

In this paper, we presented and experimentally validated a nonlinear optimization system called Sparse Pose Adjustment (SPA) for 2D pose graphs. SPA relies on efficient linear matrix construction and sparse non-iterative Cholesky decomposition

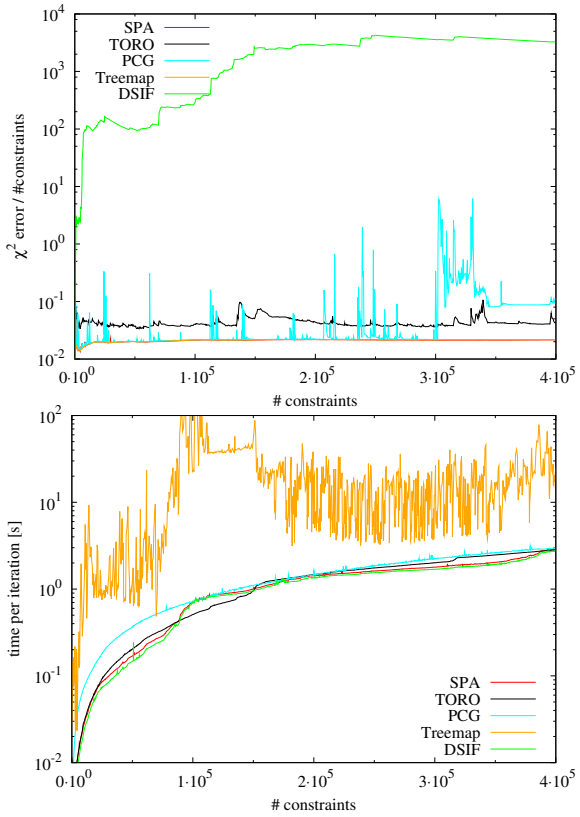

Fig. 8. On-line optimization of the large simulated data set. In the top graph, SPA and TreeMap have the same minimum χ^2 error over all constraints. DSIF does not converge to a global minimum, while TORO converges slowly and PCG spikes and has trouble with larger graphs. In the bottom figure, SPA is clearly superior to TreeMap in computation time.

to efficiently represent and solve large sparse pose graphs. None of the real-world datasets we could find were challenging – even in batch mode. The largest map takes sub-second time to get fully optimized. On-line computation is in the range of 10 ms/node at worst; unlike EKF filters or other methods that have poor computational performance, we do not have to split the map into submaps [23] to get globally minimal error.

Compared to state-of-the-art methods, SPA is faster and converges better. The only exception is in poorly-initialized maps, where only the stochastic gradient technique of TORO can converge; but by applying a spanning-tree initialization, SPA can solve even the difficult synthetic example better than TORO. When combined with a scan-matching front end, SPA will enable on-line exploration and map construction. Because it is a pose graph method, SPA allows incremental additions and deletions to the map, facilitating lifelong mapping [16].

All the relevant code for running SPA and the other methods we implemented is available online and as open-source, along with the datasets and simulation generator (`www.ros.org/research/2010/spa`). An accompanying video shows SPA in both online and offline mode on a large real-world dataset.

References

[1] M. Agrawal and K. Konolige. FrameSLAM: From bundle adjustment to real-time visual mapping. *IEEE Transactions on Robotics*, 24(5), October 2008.

[2] F. F. Campos and J. S. Rollett. Analysis of preconditioners for conjugate gradients through distribution of eigenvalues. *Int. J. of Computer Mathematics*, 58(3):135–158, 1995.

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

[4] F. Dellaert. Square Root SAM. In *Proc. of Robotics: Science and Systems (RSS)*, pages 177–184, Cambridge, MA, USA, 2005.

[5] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in c++ for high performance architectures. In *Object Oriented Numerics Conference*, pages 214–218, 1994.

[6] T. Duckett, S. Marsland, and J. Shapiro. Fast, on-line learning of globally consistent maps. *Journal of Autonomous Robots*, 12(3):287 – 300, 2002.

[7] R. M. Eustice, H. Singh, and J. J. Leonard. Exactly sparse delayed-state filters for view-based SLAM. *IEEE Trans. Robotics*, 22(6), 2006.

[8] U. Frese. Treemap: An $o(logn)$ algorithm for indoor simultaneous localization and mapping. *Journal of Autonomous Robots*, 21(2):103–122, 2006.

[9] U. Frese, P. Larsson, and T. Duckett. A multilevel relaxation algorithm for simultaneous localisation and mapping. *IEEE Transactions on Robotics*, 21(2):1–12, 2005.

[10] G. Grisetti, C. Stachniss, and W. Burgard. Non-linear constraint network optimization for efficient map learning. *IEEE Transactions on Intelligent Transportation Systems*, 10:428–439, 2009. ISSN: 1524-9050.

[11] J.-S. Gutmann, M. Fukuchi, and K. Sabe. Environment identification by comparing maps of landmarks. In *International Conference on Robotics and Automation*, 2003.

[12] J.-S. Gutmann and K. Konolige. Incremental mapping of large cyclic environments. In *Proc. of the IEEE Int. Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, pages 318–325, Monterey, CA, USA, 1999.

[13] A. Howard, M. Matarić, and G. Sukhatme. Relaxation on a mesh: a formalism for generalized localization. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, pages 1055–1060, 2001.

[14] M. Kaess, A. Ranganathan, and F. Dellaert. iSAM: Fast incremental smoothing and mapping with efficient data association. In *International Conference on Robotics and Automation*, Rome, 2007.

[15] K. Konolige. Large-scale map-making. In *Proceedings of the National Conference on AI (AAAI)*, 2004.

[16] K. Konolige and J. Bowman. Towards lifelong visual maps. In *International Conference on Intelligent Robots and Systems*, pages 1156–1163, 2009.

[17] K. Konolige and K. Chou. Markov localization using correlation. In *Proc. of the Int. Conf. on Artificial Intelligence (IJCAI)*, 1999.

[18] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Journal of Autonomous Robots*, 4:333–349, 1997.

[19] I. Mahon, S. Williams, O. Pizarro, and M. Johnson-Roberson. Efficient view-based SLAM using visual loop closures. *IEEE Transactions on Robotics*, 24(5):1002–1014, October 2008.

[20] M. Montemerlo and S. Thrun. Large-scale robotic 3-d mapping of urban structures. In *ISER*, 2004.

[21] E. Olson, J. Leonard, and S. Teller. Fast iterative optimization of pose graphs with poor initial estimates. In *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, pages 2262–2269, 2006.

[22] E. B. Olson. Real-time correlative scan matching. In *International Conference on Robotics and Automation*, pages 4387–4393, 2009.

[23] L. Paz, J. Tardós, and J. Neira. Divide and conquer: EKF SLAM in O(n). *IEEE Transactions on Robotics*, 24(5), October 2008.

[24] S. Thrun and M. Montemerlo. The graph SLAM algorithm with applications to large-scale mapping of urban structures. *Int. J. Rob. Res.*, 25(5-6):403–429, 2006.

[25] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. W. Fitzibbon. Bundle adjustment - a modern synthesis. In *Vision Algorithms: Theory and Practice*, LNCS, pages 298–375. Springer Verlag, 2000.