Fast Iterative Alignment of Pose Graphs with Poor Initial Estimates

Edwin Olson, John Leonard, and Seth Teller
MIT Computer Science and Artificial Intelligence Laboratory
Cambridge, MA 02139
Email: eolson@mit.edu, jleonard@mit.edu, teller@csail.mit.edu
http://rvsn.csail.mit.edu

***Abstract*— A robot exploring an environment can estimate its own motion and the relative positions of features in the environment. Simultaneous Localization and Mapping (SLAM) algorithms attempt to fuse these estimates to produce a map and a robot trajectory. The constraints are generally non-linear, thus SLAM can be viewed as a non-linear optimization problem. The optimization can be difficult, due to poor initial estimates arising from odometry data, and due to the size of the state space.**

We present a fast non-linear optimization algorithm that rapidly recovers the robot trajectory, even when given a poor initial estimate. Our approach uses a variant of Stochastic Gradient Descent on an alternative state-space representation that has good stability and computational properties. We compare our algorithm to several others, using both real and synthetic data sets.

I. Introduction

Many robotics problems require a robot to build a map of its environment while simultaneously determining its trajectory; this process is dubbed Simultaneous Localization and Mapping (SLAM). At its core, SLAM is an optimization problem: find the map with the greatest probability given sensor observations. Data association is a core problem in SLAM, but in this paper, we assume that data associations are known.

The SLAM optimization problem is quite difficult. It has a large search space, since the position of each pose and feature must be determined. It is also highly nonlinear, since observations are made relative to the robot's location (the constraint equations include trigonometric functions of the robot's orientation).

Perhaps the most challenging aspect of SLAM optimization is that the initial state estimate (arising from the vehicle's odometry) can be very noisy. After traveling around a large loop, the robot's estimated position may differ markedly from its true position.

Current SLAM algorithms have a limited ability to deal with poor initial estimates. The family of linearizing approaches (Extended Kalman Filter, Extended Information Filter, and the many approaches that improve upon their computational complexity [1], [2]), irrevocably introduce linearization error. We will show the effects of this linearization error.

Nonlinear optimization algorithms do not suffer from linearization error, since they can re-evaluate constraint equations as the state estimate improves. Current nonlinear methods are often slow, often failing to recover the global structure of the map in a reasonable amount of run time.

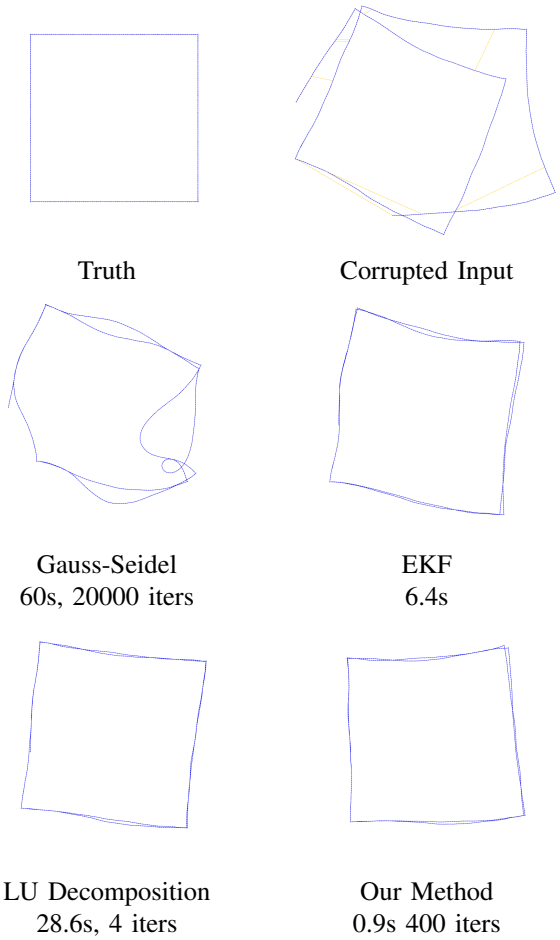


Fig. 1. SLAM experiment with about 600 poses and nine loops. Dots (very small) represent robot poses, and lines show constraints. The EKF solution exhibits noticeable degradation due to linearization error. LU Decomposition and our method both produce good results, though LU decomposition takes much longer. Notice that Gauss-Seidel has failed: an extra loop has been inserted in the lower right which will never be removed. See Fig. 2 for convergence rate plots.

We present a nonlinear map optimization algorithm which can optimize a map even when the initial estimate of the map is poor. Further, our approach is very fast. This paper makes two main contributions:

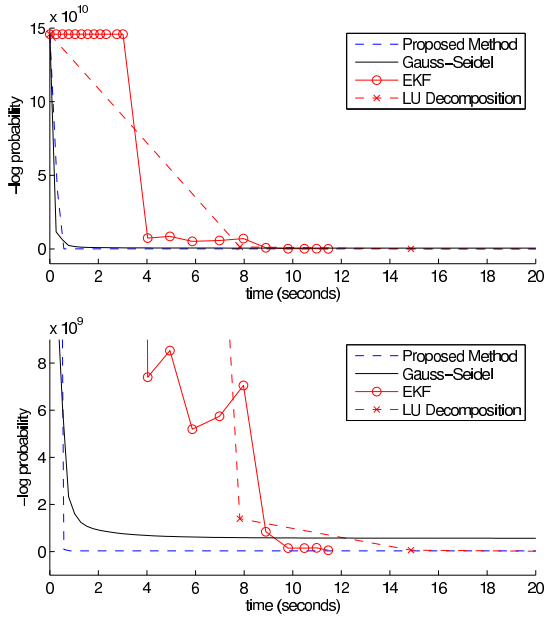


Fig. 2. Convergence rates for a simple SLAM experiment (shown in Fig. 1). Our method was dramatically faster than the others, while achieving lower error. Note that the runtime for LU decomposition was actually about 320 seconds; it has been compressed for comparison purposes. The second plot shows the same data as the first, with a different vertical scale to show detail.

- An alternative state space representation that allows a single iteration to update many poses without incurring a large computational cost.
- A variant of Stochastic Gradient Descent that is robust against local minima and converges quickly.

Our algorithm is uniquely adept at recovering the macroscopic shape of pose graphs, quickly producing maps that resemble the ground truth. Our algorithm generally does not find an exact minimum due to the approximations involved. However, given reasonable time bounds, the approximate minimum is typically better than that of other algorithms. If an exact minimum is required, our algorithm could be used to bootstrap another approach.

Our method consumes $O(N + M)$ memory for N poses and M constraints, and each iteration of the optimizer (which considers a single constraint) runs in $O(\log N)$ time. This paper demonstrates our approach on several data sets.

II. Prior Work

The SLAM problem can be represented as a graph in which nodes are features and/or robot poses, and edges are rigid-body constraints between nodes. The information matrix, the inverse of the Kalman filter's covariance matrix, has a natural interpretation as the adjacency matrix of this graph. Thrun *et al.* demonstrated that the information matrix can be effectively sparsified [3]. Eustice *et al.* showed that by tracking all robot poses, the information matrix becomes exactly sparse [4]. While information filters can rapidly incorporate new observations, exactly recovering the state estimate requires inversion of the entire information matrix. Another limitation of both Kalman and Information Filter approaches is that they irrevocably introduce linearization error when performing observation updates. When the state estimate is poor, these linearization errors can lead to poor estimates, even though the filter's covariance estimate may indicate low error.

A promising family of SLAM approaches avoids introducing permanent linearization errors by continuously relinearizing around the current state estimate. These approaches attempt to compute the fully *nonlinear* Maximum Likelihood Estimate (MLE). However, because they perform optimization in a highly non-linear domain, these approaches can suffer from local minima and unboundedly slow convergence rates.

A brute-force nonlinear least squares implementation was suggested by Lu and Milios [5]. Their approach is similar in formulation to more modern approaches, but their brute-force implementation makes their algorithm impractical.

Duckett *et al.* described an early nonlinear SLAM implementation [6] that uses Gauss-Seidel relaxation. However, they assumed absolute knowledge of the robot's orientation, essentially making the problem linear. Frese, Larsson, and Duckett addressed that limitation and attempted to improve convergence speed in [7] with the Multi-Level Relaxation (MLR) algorithm. Multi-resolution methods are typically applied to problems more spatially uniform than that of SLAM, but they report good results. MLR, given time, can generally find the exact minimum of the graph.

Other nonlinear approaches include GraphSLAM [8], and Graphical SLAM [9]. Konolige proposes a method [10] for accelerating convergence by reducing the graph to poses that have a loop constraint attached, solving for the other nodes separately. This can save considerable CPU time, but requires the graph to have low connectivity.

Paskin's Thin Junction Tree Filter [11] and Frese's TreeMap [12] compute nonlinear map estimates, but their approaches require factorization of the joint probability density, which they achieve by ignoring small state correlations. These approximations can result in noticeable map artifacts.

A hybrid of linear and nonlinear solutions is Bosse's Atlas [13], which uses linearized (EKF-based) submaps but stitches them together using nonlinear optimization.

Nonlinear optimization algorithms have a rich history outside the SLAM community. SLAM algorithms have typically limited themselves to Gauss-Seidel or Gradient Descent approaches, but other approaches are commonly used in other fields. In particular, Stochastic Gradient Descent [14] is often used to train artificial neural networks.

III. Our Method

A. Preliminaries

We consider a version of the 2D SLAM problem that explicitly optimizes only the trajectory of robot poses. As demonstrated by FastSLAM [15], positions of non-pose features can be trivially computed given the robot trajectory since they become conditionally independent. This approach reduces the search space.

In this paper, we assume that constraints between poses are always given as full-rank rigid-body transformations with uncertainty matrices. This allows us to treat constraints in a uniform way, simplifying presentation. However, rank-deficient constraints, such as those arising from data associations between lines, could be incorporated in principle. We also make the typical assumption that constraints (which arise from observations) are independent.

Before proceeding, we review how the graph can be optimized by solving a linear problem of the form $Ax = b$. If x is the state vector representing robot poses, and $f()$ represents the constraint equations with expected values u and variances Σ, we can write the log probability of the node positions as:

$$-\log\ P(x) \propto (f(x) - u)^T \Sigma^{-1} (f(x) - u) \tag{1}$$

The constraint equations are nonlinear due to effects of robot orientation. We proceed by linearizing $f(x) = F|_x + J|_x \Delta x$, using column vector $F|_x$ and matrix $J|_x$ (the Jacobian of the constraint equations with respect to the state). A single rigid-body constraint yields three constraint equations, occupying a block-row of the Jacobian.

Since we always linearize around the current state estimate, we simply write F and J. We will also write $d = \Delta x$, which is suggestive of the search *direction*, and aids readability. Finally, we also set $r = u - F$, the residual. Eqn. 1 then becomes:

$$\begin{aligned} -\log\ P(x) \quad &\propto \quad (Jd - r)^T \Sigma^{-1} (Jd - r) \\ cost \quad &= \quad d^T J^T \Sigma^{-1} J d - 2 d^T J^T \Sigma^{-1} r + \\ &\quad\ \ r^T \Sigma^{-1} r \end{aligned} \tag{2}$$

We wish to improve our map by finding a d that maximizes the probability, or equivalently minimizes the cost. Differentiating with respect to d and setting to zero, we find that:

$$(J^T \Sigma^{-1} J) d = J^T \Sigma^{-1} r \tag{3}$$

This is a linear algebra problem of the form $Ax = b$, with $A = J^T \Sigma^{-1} J$, the *information matrix*. If we solve Eqn. 3 for d once (thus limiting us to a single linearization point), we have an Extended Information Filter. (An Extended Kalman Filter would yield the same result through different means.) If we solve for d repeatedly (by re-evaluating J around the state estimate each time), we have the method of nonlinear least squares.

In the case of SLAM, the state vector is often enormous– thousands of poses, each pose with either three (2D) or six (3D) degrees of freedom. If there are 2000 poses and the A matrix is 6000×6000, a 2.8GHz Pentium-IV running MATLAB requires 127 seconds to solve Eqn. 3 just once (assuming a dense A). If there are 10000 poses, over four hours per iteration are needed.

It is often the case that d can be reasonably estimated much faster than it can be computed exactly. This is acceptable, since even an "exact" d is only the solution to a linear approximation of a non-linear problem. This motivates us to consider iterative optimization algorithms.

B. Iterative Optimization

Iterative methods approach optimization by considering only a subset of the information available in the problem. A large family of algorithms consider the curvature of the cost surface around the current estimate, ignoring the curvature farther away. These include Newton-Raphson, Gradient Descent, Conjugate Gradient Descent [16], and Simplex methods.

When the state estimate is reasonably close to the optimum, the curvature around the estimate can quickly guide these algorithms to it. However, when the state estimate is corrupted by significant noise, the local gradient may point toward a local minimum, not the global minimum. Unless gradient methods happen to stumble out of the local basin, they will typically fail to achieve a satisfactory solution.

For many SLAM problems, the cost function can have long valleys where the gradient is nearly zero. These arise, for example, when two clusters of nodes are only loosely coupled to each other: so long as the individual clusters are well-aligned, the clusters can rotate with respect to each other with very little impact on the total cost. In this situation, some methods can converge so slowly that they appear to be stuck in a local minimum.

Another family of iterative approaches considers only a subset of the graph's nodes and edges. Gauss-Seidel, for example, considers a single node p and its edges. Fixing p's neighbors, the location of p is recomputed. Gauss-Seidel has a major shortcoming: if a cluster of nodes are far from their optimal position, but are positioned correctly *relative* to each other, they will not be moved. Since only those nodes which border an error are adjusted, it takes many iterations for the effect of a loop closure to be percolated around the graph.

The basic idea of Stochastic Gradient Descent (SGD) is to consider a single edge and the gradient with respect to just that edge. The state is then moved in the direction of greatest descent. It is "stochastic" in the sense that the constraint is usually selected randomly.

Typically, different edges will lead to steps in different directions. SGD thus tends to hop around from one local minimum to another. SGD is also less likely to be caught in a long valley, since there is probably at least one edge which has a significant gradient. This edge will cause the state estimate to teleport to another part of the cost function, where the gradient may be more helpful.

The distance that SGD travels for each edge is slowly decreased over time in order prevent oscillation. This makes it increasingly difficult for the state estimate to transition from one local minimum to another, with the result that it becomes increasing likely that SGD will get stuck in the most popular minimum (which is likely the global minimum). The degree to which SGD modulates its step size is known as the *learning rate*, and is analogous to the cooling rate in simulated annealing methods.

A great number of *learning rate schedules* have been explored in the literature ([17] is an interesting example); the simplest are simple functions of iteration number, while others

incorporate convergence rate and other information. The simplest strategy is to set the learning rate $\alpha \propto 1/n$, where n is the current iteration of the algorithm. The proportionality constant is determined, in our case, by examining the "stiffnesses" of the poses, as will be described in section III-E.

C. Derivation

Using some of the ideas underlying Stochastic Gradient Descent, we now derive our method. We begin by considering the cost of a single constraint i and its residual r_i, evaluated at the current state (i.e., the i^{th} term of Eqn. 2 with $d = 0$):

$$c_i = r_i^T \Sigma_i^{-1} r_i \tag{4}$$

The gradient of the cost function with respect to d (the change in state) can be found by applying the chain rule to Eqn. 4. With J_i the Jacobian of the i^{th} constraint with respect to d, we have:

$$\begin{aligned} \nabla c_i = \frac{\partial c_i}{\partial \overrightarrow{d}} &= \frac{\partial c_i}{\partial \overrightarrow{r_i}} \frac{\partial \overrightarrow{r_i}}{\partial \overrightarrow{d}} \\ &= 2 r_i^T \Sigma_i^{-1} J_i \end{aligned} \tag{5}$$

Stochastic Gradient Descent would then prescribe that we set $d = \alpha \nabla c_i^T$, where α represents the learning rate. (The transpose arises from the convention of gradients being row vectors and state vectors being column vectors.) In other words:

$$d = 2\alpha J_i^T \Sigma_i^{-1} r_i \tag{6}$$

However, note that summing ∇c_i^T for all constraints yields the right-hand side of Eqn. 3, up to a scale factor. Consider the left-hand side and suppose that we find an easily invertible matrix M such that $M \approx J^T \Sigma^{-1} J$ (note that J is the *full* Jacobian, not just the Jacobian for the i^{th} constraint). We can then write d so that it more accurately approximates the solution to Eqn. 3:

$$d \approx 2\alpha M^{-1} J_i^T \Sigma_i^{-1} r_i \tag{7}$$

A natural choice is to pick the diagonal matrix M with the same diagonal elements as $J^T \Sigma^{-1} J$. This is also known as Jacobi Preconditioning [18].

Thus far, our method can be viewed as incrementally building up a solution to Eqn. 3, one constraint at a time. Note that we propose to systematically iterate through constraints, rather than selecting them randomly as would Stochastic Gradient Descent.

We now make an additional modification, motivated by the fact that we know *a priori* the value of d which exactly satisfies the constraint. While Stochastic Gradient Descent can overshoot the minimum, we use this knowledge to clamp d so that we never overshoot. In general, clamping will only occur when the learning rate is large.

D. State Space Representation

Our discussion thus far has made no assumptions about which state space representation is used. However, the choice of state space has a large impact on the ultimate performance of the algorithm.

The most widely used state space is *global pose*. The global pose representation is simply the set of (x, y, θ) coordinates for each pose. With this representation, the Jacobian of a single rigid-body constraint is sparse. Usually this is a desirable thing (since it permits efficient computation), but the sparsity also means that only the two poses adjacent to the edge are directly affected by the constraint. This is *undesirable* because the motion of the robot is cumulative: moving one pose typically has an affect on a large number of other poses. Naturally, this can be addressed by considering many constraints simultaneously, but the resulting system rapidly becomes more difficult to solve.

A second possible state space is the set of rigid-body transformations that relate successive poses, which we call *relative pose*. The pose of the i^{th} pose is the composition of the first $(i-1)$ rigid-body transformations. Each of these rigid-body transformations is parameterized by three quantities: a forward translation, a sideways translation, and a relative rotation. A single constraint between two poses now affects many poses, since the relative position of two poses is a function of all of the rigid-body transformations between them. Consequently, each iteration will move a number of poses, permitting larger steps towards the minimum. However, the Jacobians that result are highly non-linear (due to the effects of each pose's orientation), and non-sparse (since many rigid-body transformations are used).

We propose using a state representation, *incremental pose*, that captures the cumulative nature of the robot's motion, but has a particularly convenient structure allowing efficient computation. The state space is composed of the incremental change in robot pose between two successive poses, i.e., the difference between them. If the global robot poses are (x_i, y_i, θ_i), the state vectors for relative and incremental pose are written:

$$\underbrace{\begin{bmatrix} cos(\theta_0)(x_1 - x_0) + sin(\theta_0)(y_1 - y_0) \\ -sin(\theta_0)(x_1 - x_0) + cos(\theta_0)(y_1 - y_0) \\ \theta_1 - \theta_0 \\ cos(\theta_1)(x_2 - x_1) + sin(\theta_1)(y_2 - y_1) \\ -sin(\theta_1)(x_2 - x_1) + cos(\theta_1)(y_2 - y_1) \\ \theta_2 - \theta_1 \\ cos(\theta_2)(x_3 - x_2) + sin(\theta_2)(y_3 - y_2) \\ -sin(\theta_2)(x_3 - x_2) + cos(\theta_2)(y_3 - y_2) \\ \theta_3 - \theta_2 \\ \dots \end{bmatrix}}_{\mathbf{x_{rel}}} \quad \underbrace{\begin{bmatrix} x_0 \\ y_0 \\ \theta_0 \\ x_1 - x_0 \\ y_1 - y_0 \\ \theta_1 - \theta_0 \\ x_2 - x_1 \\ y_2 - y_1 \\ \theta_2 - \theta_1 \\ \dots \end{bmatrix}}_{\mathbf{x_{incr}}}$$

In the x_{incr} state space, rather than having to compose a number of rigid-body transformations in order to compute the relative position between two nodes, we simply add a number of incremental motions. The Jacobian of a constraint between

pose 0 and pose 2, for example, is just:

$$J_i = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \dots \end{bmatrix} \tag{8}$$

With such a simple structure, J need never be explicitly constructed or stored.

In other words, our method considers a single constraint at a time. This constraint forms a loop with respect to the robot trajectory that we are optimizing: the sum of the motions along a portion of the trajectory must match the constraint. We compute the residual of the constraint, and the Jacobian tells us how to modify the robot trajectory to better satisfy the constraint. In essence, we "spread" the error in the loop constraint out over the trajectory. This generally reduces our total error, since many small errors cost less than one large error (due to the quadratic nature of Eqn. 2).

Intuitively, we would expect that the error should *not* be distributed evenly: the position of some nodes may be more constrained than others. This is precisely the effect of premultiplying by M^{-1}. Each constraint links two poses, adding tension to the portion of the trajectory between those poses. The "stiffness" of any particular node is the sum of all the tensions which affect it. (The "tension" of the constraints are the block-diagonals of Σ^{-1}, and the product $J^T\Sigma^{-1}J$ performs the summing.)

Eqn. 7 multiplies the Jacobian by the weighted residual. Note, however, that the rows in the Jacobian are not unit norm: constraints connecting more distant nodes have more 1's in them. In effect, this amplifies the effect of constraints according to the length of their loop. This is desirable since the error in a large loop can be spread out over many nodes; intuitively, larger loops cost less to satisfy, and *should* be given more weight. Our clamping heuristic prevents this behavior from resulting in inappropriately large steps while α is large.

E. Learning Rate

The learning rate α controls the convergence properties of stochastic gradient descent. If α decreases too quickly, the gradient steps will become small before a good solution is found. Conversely, if α decreases very slowly, convergence will be needlessly delayed (though the ultimate solution will likely be good).

We use a typical (though simple) learning rate suggested by Robbins and Monro [14]. Given the current iteration number t and a parameter γ:

$$\alpha = 1/(\gamma t) \tag{9}$$

This schedule permits large steps early on (correcting errors on a macroscopic scale and permitting the graph to visit local minima), while gradually reducing the step size in later iterations.

If all of the constraint covariances are scaled by a constant factor, the graph has the same solution (the constant factor appears on both sides of Eqn. 3). Parameter γ gives us an opportunity to account for this scaling. Consequently, we propose:

$$\gamma = min_i\ \Sigma_i^{-1} \tag{10}$$

This learning schedule is likely suboptimal, though it appears to perform well over a variety of problems. "Search-then-converge" is a slightly more sophisticated schedule that changes the expression for α so that it stays near 1 for a longer initial search period [19]. Adaptive rate schedules could provide an even larger advantage, though we have not explored them.

F. Implementation

In a practical implementation, much of the algebraic complexity vanishes once the properties of the simple Jacobian are fully exploited. In this section, we describe our geometric world model, some notation that is handy in this domain, and provide a more practical algorithmic description to aid those wishing to implement our approach. Not including a matrix library, a simple version of our method can be implemented in about 80 lines of Java code.

Each constraint is parameterized by two pose numbers (a and b), a rigid-body transformation relating them (t_{ab}), and the rigid-body transformation's covariance matrix, Σ_{ab}.

A rigid-body transformation in 2D has three parameters, δx, δy, and $\delta\theta$; we represent it with a 3×1 vector t, corresponding to a 3×3 transformation matrix T. We will freely switch back and forth between the two representations:

$$t_i = \begin{bmatrix} \delta x_i \\ \delta y_i \\ \delta\theta_i \end{bmatrix} \Longleftrightarrow T_i = \begin{bmatrix} \cos(\delta\theta_i) & -\sin(\delta\theta_i) & \delta x_i \\ \sin(\delta\theta_i) & \cos(\delta\theta_i) & \delta y_i \\ 0 & 0 & 1 \end{bmatrix} \tag{11}$$

Vehicle poses are expressed as rigid body transformations (p_i and its dual P_i) relative to the origin. For example, if the robot is at p_i and observes a feature two meters in front of it, the position of that feature in the global frame is $P_i[2\ 0\ 1]^T$ (with the vector written in homogenous coordinates.) We also write $R = Rot(P)$ to set R to just the rotational component of rigid-body transformation P, i.e., where $R_{1,3} = 0$ and $R_{2,3} = 0$. This is useful when projecting a covariance matrix into a different frame.

Each pose has a state vector x_i, which is the difference in absolute poses between it and the prior pose.

$$x_i = p_i - p_{i-1} \tag{12}$$

Note that we fix $x_0 = p_0 = [0\ 0\ 0]^T$. Consequently:

$$p_i = \sum_{j\in[0,i]} x_j \tag{13}$$

The structure of the Jacobian makes it possible for us to avoid constructing J or d explicitly. Instead, we compute the weighted residual $M^{-1}\Sigma^{-1}r$, clamp it, then add it to the appropriate state elements.

The straightforward method of adding the weighted residual to the state elements runs in $O(N)$ time (for a graph with N

poses). This approach is easy to implement, and is shown in Alg. 1.

However, it is possible to perform this operation in $O(\log N)$ time. Factoring out M, all of the affected elements of x_{incr} move by the same amount. Note that the affected elements of x_{incr} are always a contiguous portion of the trajectory.

Consider a binary tree whose leaves are the elements of x_{incr}. As we iterate over the graph's constraints, we compute changes to the elements of x_{incr}; this tree stores and sums those changes. If we define the total change for a leaf to be the sum of all the tree nodes between it and the root, we can add a change to any contiguous subset of x_{incr} in $O(\log N)$ time. Reading a value of x_{incr} becomes $O(\log N)$ as well. The per-pose scaling is implemented by scaling the results read from the tree by M_i^{-1}, and by scaling the changes that are inserted into the tree by the total weight of the poses in each loop.

Importantly, the cost of computing a pose does not rise to $O(N)$; they can be computed in $O(\log N)$, by employing a similar tree. While fairly straightforward, the implementation is rather tedious and we omit it here. An implementation is available on the author's website.

Also note that it is unnecessary to recompute M at every iteration since the rough alignment of the map is generally determined in the first few iterations. In our performance-conscious implementation, we recompute M only at iterations 1, 2, 4, 8, and so on.

Our algorithm uses $O(N + M)$ memory, for N states and M constraints. The coefficients for each are small: neglecting language overhead, constraints consume 80 bytes each, and poses require 24 bytes. A graph with a million poses and two million constraints would require about 160MB (though we have not tried such a large problem).

Since each constraint can be evaluated in $O(\log N)$, the total time per iteration is $O(M \log N)$. However, we do not claim that our algorithm *converges* in $O(M \log N)$ time (i.e., in a constant number of iterations). In fact, defining "convergence" is tricky: particular choices of the learning-rate parameter γ can force the step size to be small in a predictable amount of time, but this does not imply that the solution is near a minimum. This is a problem common to stochastic optimization [19], and is a subject for future work.

Algorithm 1 SGD Optimize Graph

iters = 0
loop
  iters++
  $\gamma = [\infty\ \infty\ \infty]^T$

  {Update approximation $M = J^T\Sigma^{-1}J$}
  $M = \text{zeros}(numstates, 3)$
  for all {a, b, t_{ab}, Σ_{ab}} in $Constraints$ **do**
    $R = Rot(P_a)$
    $W = (R\Sigma_{ab}R^T)^{-1}$
    for all $i \in [a+1, b]$ **do**
      $M_{i,1:3} = M_{i,1:3} + diag(W)$
      $\gamma = \min(\gamma, diag(W))$

  {Modified Stochastic Gradient Descent}
  for all {a, b, t_{ab}, Σ_{ab}} in $Constraints$ **do**
    $R = Rot(P_a)$
    $P_b{}' = P_a T_{ab}$
    $r = p_b{}' - p_b$
    $r_3 = mod2pi(r_3)$
    $d_{1:3} = 2(R^T\Sigma_{ab}R)^{-1}r$

    {Update x, y, and θ}
    for all $j \in [1, 3]$ **do**
      $alpha = 1/(\gamma_j * iters)$
      $totalweight = \sum_{i\in[a+1,b]} 1/M_{i,j}$
      $\beta = (b-a)d_j\alpha$
      if $|\beta| > |r_j|$ **then**
        $\beta = r_j$
      $dpose = 0$
      for all $i \in [a+1, N]$ **do**
        if $i \in [a+1, b]$ **then**
          $dpose = dpose + \beta/M_{i,j}/totalweight$
        $p_{i,j} = p_{i,j} + dpose$

IV. Results

We present results on two synthetic problems, a simple double-loop and a very complex environment, and also on real data from the large Killian Court data set.

Synthetic experiments on randomly-generated graphs allow us to explore our algorithm's behavior on extremely large environments for which we do not have real datasets. It also allowed us to test our algorithms with many different noise characteristics, in order to be able to confidently say that the results presented here are typical.

We note that our graph generator creates "grid-world" graphs, as though a robot was traveling the city streets in upper Manhattan. This makes it easy to visually appraise a map: corners which appear to be almost ninety degrees *should* be ninety degrees. Paths which appear to overlap actually do. The graph optimizers were in no way tuned to exploit this property.

A. Double Loop

On the double-loop experiment (see Fig. 1), the graph size was small enough to allow us to try many different algorithms, including those with $O(N^2)$ or worse complexity.

LU decomposition (nonlinear least squares) converged in four iterations to a very good solution, but required over 28 seconds of CPU time. The EKF's performance is tolerable, though it is noticeably worse than the nonlinear solutions.

Gauss-Seidel struggles with large worlds due to its adjustment of only one node at a time. Nodes that are close to a loop closure very quickly move into relative alignment, but the amount of adjustment decreases very quickly with distance. In other words, it is often the case that the majority

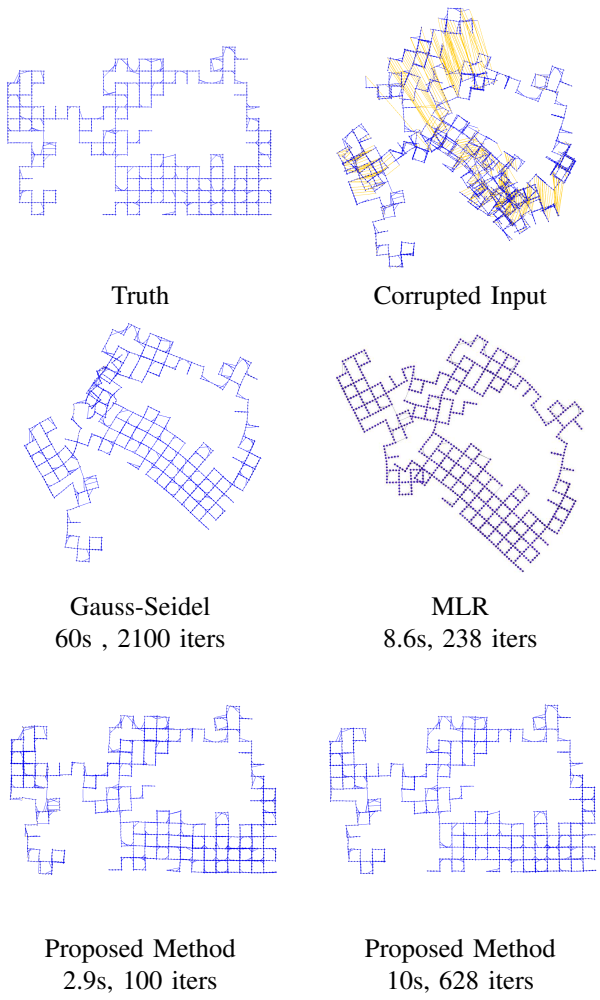


Fig. 3. Difficult SLAM problem consisting of 3500 poses and 5600 constraints. MLR and Gauss-Seidel have dramatically reduced the cost function, but their solutions are unsatisfactory. Given 1833 seconds, MLR converges to a satisfactory solution. Convergence rates are shown in Fig. 4.

of the "tension" in the graph can be released by adjusting comparatively few nodes. Once the tension is reduced, the steps become quite small, even if the graph is far from the optimum.

Another consequence of Gauss-Seidel relaxation is that nodes bordering large errors can move dramatically. In this experiment, a number of nodes (in the lower right) jumped a large distance, inducing an extraneous loop into the graph. Based on our experiments, we do not believe this loop will ever be corrected: Gauss-Seidel has become trapped in a local minimum.

Our proposed method reduces the error much faster than the other three algorithms (see Fig. 2). As expected, LU Decomposition ultimately converges to the lowest error (-8.09×10^4). Our approach (-2.75420×10^7) outperforms the EKF by a smaller but still significant margin (-5.70×10^7). Gauss Seidel, after 60 seconds of CPU time, is the worst (-5.63×10^8).

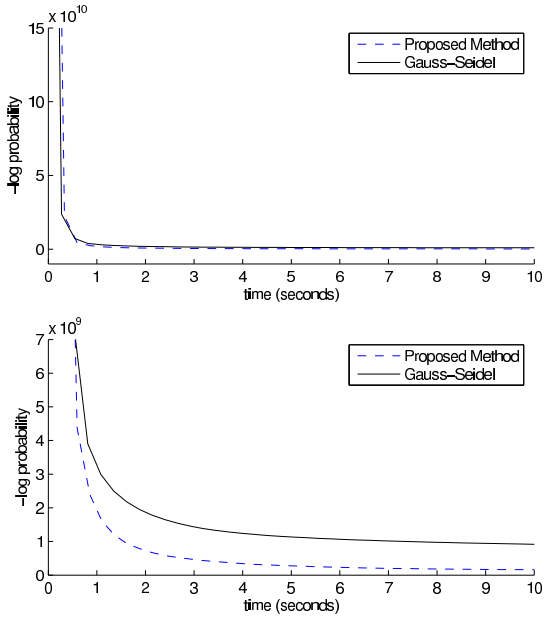


Fig. 4. Convergence rates for problem in Fig. 3. Our method converges much faster than Gauss-Seidel and produces a higher quality map. MLR (not shown) reduces the cost function faster than our method, but the resulting graphs do not resemble the ground truth (see Fig. 3)

B. Complex Graph

In order to demonstrate our algorithm on a difficult problem, we generated a large graph with significant error in both the initial state and loop closures. The graph consists of 3500 poses and 5600 constraints. In this graph, the robot visits different areas with highly variable frequencies; some areas are visited up to 13 times, while others are visited only once (the mean was about 2).

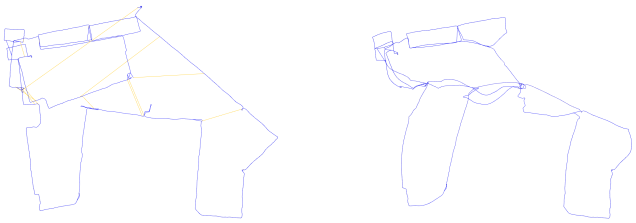
Since in 2D there are three degrees of freedom, the Jacobian (were it to be computed explicitly) would be 16800×10500. The size of this problem makes it impossible to run LU-Decomposition or an EKF implementation.

Our proposed method outperformed Gauss-Seidel both in terms of convergence rate and subjective graph quality. On this data set, we have results using Multilevel Relaxation (MLR) [7]. MLR's convergence rate (log probability) is actually substantially faster than our method, but the subjective quality of the maps is only marginally better than Gauss-Seidel (see Fig. 3). After 1800 seconds of CPU time, MLR converges to a minimum that corresponds to a subjectively high-quality map.

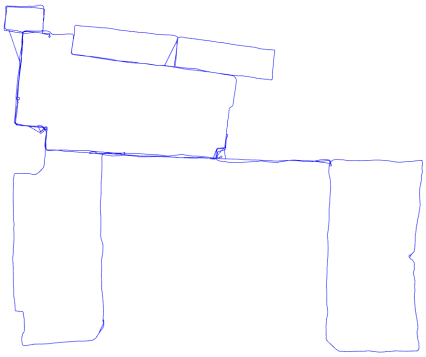
These results illustrate that log probability is not an adequate measure of graph quality. This is an area for future study.

C. Killian Court

The Killian Court dataset is one of the larger SLAM datasets, which after processing with our laser scan code, produced about 1900 poses. Laser scan matching provided the constraints connecting consecutive poses. Twenty additional loop closures were identified manually by aligning laser scans.



Corrupted Input Gauss-Seidel (60s)



Proposed Method (1s)

Fig. 5. Killian Court. The Killian data set is composed of about 1900 poses, with a path length of 1,450 meters. Two hours of robot travel resulted in a pose graph that was optimized in under a second using our algorithm.

Fig. 6. B21 Robot. This B21 was used to collect the Killian data set. Laser scan data was processed offline to produce a pose graph. This pose graph was then optimized by our algorithm.

Our algorithm rapidly recovered the structure of the environment (in about 1 second), while Gauss-Seidel made poor progress given 60 seconds. These results demonstrate the applicability of our algorithm to real-world datasets.

V. Conclusion

We have presented a fast iterative algorithm for optimizing pose graphs, even in the presence of significant open-loop error. It estimates the maximum likelihood solution, often achieving lower error than a full EKF solution, and converges much faster than most other approaches while producing maps that closely resemble the true state.

The robustness and speed of our approach make it compelling as a stand-alone SLAM algorithm, but it could also be used by other non-linear algorithms like GraphSLAM or MLR to improve their convergence on difficult problems by providing a better initial estimate.

VI. Acknowledgements

We thank Udo Frese for helpful discussions regarding the $O(M \log N)$ variant of the algorithm, and for running his MLR code on our datasets. Also, we thank Mike Bosse for collecting and sharing the Killian Court data set.

This material is based upon work supported by the National Science Foundation under Grant No. 0514639

References

[1] J. Knight, A. Davison, and I. Reid, "Towards constant time SLAM using postponement," 2001. [Online]. Available: citeseer.ist.psu.edu/article/knight01towards.html

[2] J. Guivant and E. Nebot, "Compressed filter for real time implmentation of simultaneous localization and map building," *FSR 2001 International Conference on Field and Service Robots*, 2001.

[3] S. Thrun, Y. Liu, D. Koller, A. Ng, Z. Ghahramani, and H. Durrant-Whyte, "Simultaneous localization and mapping with sparse extended information filters," April 2003.

[4] R. Eustice, H. Singh, and J. Leonard, "Exactly sparse delayed-state filters," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, Barcelona, Spain, April 2005, pp. 2428–2435.

[5] F. Lu and E. Milios, "Globally consistent range scan alignment for environment mapping," 1997. [Online]. Available: citeseer.nj.nec.com/lu97globally.html

[6] T. Duckett, S. Marsland, and J. Shapiro, "Learning globally consistent maps by relaxation," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'2000)*, San Francisco, CA, 2000.

[7] U. Frese, P. Larsson, and T. Duckett, "A multilevel relaxation algorithm for simultaneous localisation and mapping," *IEEE Transactions on Robotics*, 2005.

[8] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2005.

[9] J. Folkesson and H. I. Christensen, "Graphical SLAM - a self-correcting map," *Proceedings of the IEEE International Conference on Robotics and Automation*, pp. 791–798, 2004.

[10] K. Konolige, "Large-scale map-making." in *AAAI*, 2004, pp. 457–463.

[11] M. Paskin, "Thin junction tree filters for simultaneous localization and mapping," Ph.D. dissertation, Berkeley, 2002.

[12] U. Frese, "Treemap: An $O(log(n))$ algorithm for simultaneous localization and mapping," in *Spatial Cognition IV*, 2004.

[13] M. Bosse, P. Newman, J. Leonard, and S. Teller, "An Atlas framework for scalable mapping," vol. 23, no. 12, pp. 1113–1139, December 2004.

[14] H. Robbins and S. Monro, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 1951.

[15] M. Montemerlo, "FastSLAM: A factored solution to the simultaneous localization and mapping problem with unknown data association," Ph.D. dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, July 2003.

[16] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Pittsburgh, PA, USA, Tech. Rep., 1994.

[17] N. N. Schraudolph, "Local gain adaptation in stochastic gradient descent, Tech. Rep. IDSIA-09-99, 8, 1999.

[18] G. Strang, *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 1993.

[19] C. Darken, J. Chang, and J. Moody, "Learning rate schedules for faster stochastic gradient search," in *Proc. Neural Networks for Signal Processing 2*. IEEE Press, 1992. [Online]. Available: citeseer.ist.psu.edu/darken92learning.html