Spatially-Adaptive Learning Rates for Online Incremental SLAM

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*— Several recent algorithms have formulated the SLAM problem in terms of non-linear pose graph optimization. These algorithms are attractive because they offer lower computational and memory costs than the traditional Extended Kalman Filter (EKF), while simultaneously avoiding the linearization error problems that affect EKFs.**

In this paper, we present a new non-linear SLAM algorithm that allows incremental optimization of pose graphs, i.e., allows new poses and constraints to be added without requiring the solution to be recomputed from scratch. Our approach builds upon an existing batch algorithm that combines stochastic gradient descent and an incremental state representation. We develop an incremental algorithm by adding a spatially-adaptive learning rate, and a technique for reducing computational requirements by restricting optimization to only the most volatile portions of the graph. We demonstrate our algorithms on real datasets, and compare against other online algorithms.

I. Introduction

Simultaneous Localization and Mapping (SLAM) algorithms compute a map of an environment using feature observations and estimates of robot motion. SLAM can be viewed as an optimization problem: find a configuration of features and a robot trajectory that is maximally probable given the *constraints* (the sensor observations and robot motion estimates).

The Kalman filter (and its dual, the information filter) are classical approaches to the SLAM problem that assume that the map estimation problem is *linear*, i.e., that uncertainties can be modeled as Gaussians and that the constraint equations are linear in the state variables. Neither is true in practice, but the resulting approximations permit closed-form optimization of the posterior using straight-forward linear algebra. While these algorithms are simple in *form*, their run-time and memory requirements increase quadratically in the number of poses. Many authors have attempted to address these costs [1, 2]. The Iterated [3] and Unscented filters [4] improve the performance of these classical filters in non-linear domains.

Particle filter approaches like FastSLAM [5] explicitly sample the posterior probability distribution, allowing any distribution to be approximated. Unfortunately, large numbers of particles are required to ensure that an acceptable posterior estimate is produced. Supporting large particle populations leads to computational and memory consumption issues.

Non-linear constraints can also be handled by iteratively updating an estimate, each time linearizing around the current

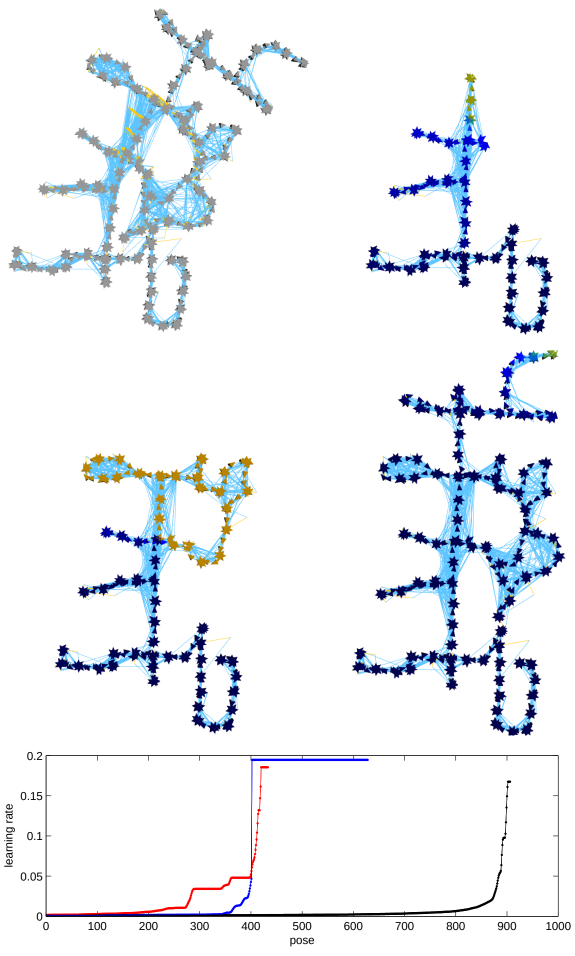


Fig. 1. Incremental Processing of Freiburg dataset. The open-loop graph (top-left) is incrementally optimized; the state of the graph is shown at two intermediate configurations and the final configuration. The colors used in the map indicate the learning rates Λ_i, which are also plotted on the bottom. When closing large loops (middle-left figure), the learning rate is increased over a larger portion of the graph.

state estimate. Lu and Milios suggested a brute-force method [6] that is impractical for all but small problems. Sparse factorizations of the information matrix permit faster updates; TreeMap [7] and Thin Junction Tree Filters [8] truncate small values to enable efficient factorizations, while Square-root SAM searches for a variable reordering that produces sparse but still exact factorizations [9].

Maps with non-linear constraints can also be iteratively improved without computing a factorization of the information matrix. Dellaert proposed a simple relaxation based scheme [10], which was improved by Frese [11]; both of these methods iteratively improve the state variables (the poses), considering a subset of them at a time. More recently, we proposed an alternative method [12] similar to stochastic gradient descent [13]; this method approaches optimization by considering the constraints, rather than the poses. However, these algorithms are *batch* algorithms, and thus are not well-suited to online use.

In this paper, we develop an incremental non-linear optimization algorithm extending our previous batch algorithm [12]; namely, the method is based upon stochastic gradient descent operating on the incremental state representation. The central contributions of this paper are:

- An on-line (incremental) variation of an algorithm that could previously only be used in batches;
- A spatially-varying learning rate that allows different parts of the graph to converge at different rates in correspondence with the impact of new observations;
- A method for accelerating convergence by iterating only on the most volatile parts of the graph, reducing the number of constraints that need to be considered during an iteration.

Iterative methods, like the one described in this paper, are well suited for on-line use: they can incorporate new observations very quickly, and can produce successively better posterior estimates using as much or as little CPU time as the robot can afford. Because the CPU requirements can be throttled, and because the memory requirements are linear in the number of poses and constraints, our approach is well-suited to computationally constrained robots. Data association algorithms also benefit from online algorithms, as the partial maps they produce can be used to help make new associations.

II. Previous Method

This section briefly reviews the batch optimization algorithm described in [12]. The algorithm takes modified gradient steps by considering a single constraint at time. An alternative state space representation is also employed.

Consider a set of robot poses x and a set of constraints that relate pairs of poses. Let J_i be the Jacobian of the i^{th} constraint, and J be the Jacobian of all constraints. Similarly, let Σ_i^{-1} be the information matrix for the i^{th} constraint, and r_i the i^{th} residual. In this paper, we assume that constraints are rigid-body transformations (though generalizations are possible): this means that if there are C constraints and N poses, J will be $3C \times 3N$ and Σ^{-1} will be $3C \times 3C$. The factors of three reflect the degrees-of-freedom inherent in a 2D rigid-body transformation (translation in $\hat{x}$, $\hat{y}$, and rotation).

Given some small step d from the current state estimate, we can write the χ^2 error for all the constraints as:

$$\chi^2 = (Jd - r)^T \Sigma^{-1} (Jd - r) \quad (1)$$

Differentiating with respect to d results in the normal equations for the system:

$$J^T \Sigma^{-1} J d = 2 J^T \Sigma^{-1} r \quad (2)$$

Note that solving this expression for d would yield a least-squares iteration. Now, considering the effects of a single constraint i (i.e., setting $r_j = 0$ for all $j \neq i$), we obtain the step:

$$d = (J^T \Sigma^{-1} J)^{-1} J_i^T \Sigma_i^{-1} r_i \quad (3)$$

This expression cannot be easily evaluated, as it requires the inversion of the information matrix. The quantity $J_i^T \Sigma_i^{-1} r_i$ corresponds to the pure gradient step: the inverted information matrix can be interpreted as a weighting term that accelerates convergence by incorporating knowledge of the relative importance of other constraints on each state variable.

We can accelerate convergence versus a pure gradient step by approximating the information matrix with a matrix M. As in [12], we use the diagonal elements of the information matrix (which are easily computed). This approximation is coarse; in partial compensation we scale all matrix-vector products such that the magnitude of the resulting vector is the same as the original vector. In other words, we use the *shape* of M, but use the magnitude of a gradient-descent step.

The approach in [12] also employed a novel state representation that leads to Jacobians with a particularly simple form that permits fast updates. For each pose, the three unknowns are rewritten as the sum of global-relative increments. Each variable (x, y, and θ) is handled independently; for example:

$$x_i = \sum_{j=0}^{i-1} \Delta x_j \quad (4)$$

This change of variables is motivated by the fact that robot motion is cumulative: the position of a given pose is a function of the motions that preceded it. This could also be accomplished by using rigid-body transformations as the state variables, but the incremental representation leads to a particularly simple Jacobian whereas rigid-body motions lead to complex Jacobians.

Consider a constraint connecting poses a and b, which is a function of the motions between a and b. The Jacobian is well-approximated by zero for the poses between $[0, a]$, block-wise constant for the poses $[a + 1, b]$, and zero for the poses after b. This special structure allows a step to be taken in $O(\log N)$ time, as described in [12].

As with stochastic gradient descent, a learning rate λ is employed with each step. Without a learning rate, antagonistic constraints would cause the state estimate to forever oscillate; the learning rate allows these constraints to find an equilibrium by forcing them to compromise. Over time, the learning rate is decreased according to a harmonic progression, the standard rate schedule for stochastic gradient descent [13].

Gradient steps are scaled by the magnitude of the covariance matrix, but the maximum likelihood solution is affected only by their *relative* magnitudes: this results in different convergence behavior for problems differing only by a scale factor. In a least-squares iteration, the correct scaling is determined via inversion of the information matrix, but in our case, this is too costly to compute (and our estimate M is far too coarse). We can, however, rescale the problem such that the magnitudes of the covariance matrices are approximately 1; we write this scale factor as Ω. The parameter Ω is not critical; the average value of Σ_i is generally a reasonable choice.

Combining all of these elements, the step size used in [12] can be written:

$$d_i = \lambda \Omega M^{-1} J_i^T \Sigma_i^{-1} r_i \tag{5}$$

Recall that the scaling by M^{-1} is really a more complicated operation that preserves the amount by which the residual will be reduced. In [12], Eqn. 5 is implemented by constructing a binomial tree from the scaling weights M, then distributing the total *residual reduction* over its leaves (the poses). Consequently, we actually need to calculate the total reduction in residual, Δr_i that results from adding d_i to the state estimate.

Because M^{-1} preserves the residual reduction, Δr_i is independent of M^{-1}. Recall that the Jacobian J_i is well-approximated as zero, except for a block matrix that is repeated $(b-a)$ times. The repeated matrix is in fact a rotation matrix, which we will call R. Multiplying out Eqn. 5 and summing the incremental motion between each pose, we can compute Δr_i:

$$\Delta r_i = \lambda (b-a) \Omega R \Sigma_i^{-1} r_i \tag{6}$$

If necessary, we clamp Δr_i to r_i, to avoid stepping past the solution. Stepping past *might* result in faster convergence (as in the case of successive over-relaxation), but increases the risk of divergence.

This method can rapidly optimize graphs, even when the initial state estimate is poor. This robustness arises from considering only one constraint at a time: the large noisy steps taken early in the optimization allow the state estimate to escape local minima. However, once the solution lands in the basin of the global minimum, the constraints tend to be well-satisfied and smaller steps result.

However, as described above and in [12], the algorithm operates in *batch* mode: new poses and constraints cannot be added to the graph once optimization begins. This makes the algorithm poorly suited for online applications.

III. Incremental Extension

A. Overview

This paper presents a generalization of the batch algorithm that allows new poses and new constraints to be added without restarting the optimization.

When adding a new constraint to a graph, it is desirable to allow the state estimate to reflect the new information fairly quickly. This, in general, requires an increase in the learning rate (which can otherwise be arbitrarily small, depending on how many optimization iterations have been performed so far).

However, large increases in the learning rate cause large steps, obliterating the fine-tuning done by previous iterations. The challenge is to determine a learning rate increase that allows a new constraint to be rapidly incorporated into the state estimate, but that also preserves as much of the previous optimization effort as possible.

Intuitively, a good approach would have the property that a constraint that contained little new information would result in small learning rate increases. Conversely, a new constraint that radically alters the solution (i.e., the closure of a large loop) would result in a large learning rate increase.

When new constraints are added to a graph, their effects are often limited to only a portion of the graph. A good approach should insulate stable parts of the graph from those parts that are being reconfigured due to the addition of new constraints.

To be worthwhile, any candidate approach must be faster than the batch algorithm. It would also be compelling if the incremental algorithm was equivalent to the batch algorithm when the set of constraints is fixed.

This section describes our approach, which has the desirable properties outlined above. Note that we do not discuss how graph constraints are computed (or where they come from): we assume that they are produced by some external sensor system, such as a laser scan-matching algorithm [14] or vision system [15].

B. Spatially-varying learning rates

It is desirable to be able to insulate one area of the graph from the effects of another area of the graph. Suppose that a robot is traveling in a world with two buildings: first it explores building A, then building B. Suppose that the robot discovers that two rooms are in fact the same room in building B: we intuitively expect that the map of building B might require a substantial modification, but the map of building A should be virtually unchanged.

If a significant reconfiguration of building B needlessly causes a violent reconfiguration of building A, the optimization effort previously expended to optimize the map of building A would be wasted. This is to be avoided.

We can isolate one part of the graph from other parts of the graph by spatially varying the learning rate. Instead of a global learning rate λ, we give each pose a different learning rate Λ_i. This allows the learning rate to be varied in different parts of the graph. Managing these learning rates is the subject of this section.

C. Adding a new constraint

When adding a new constraint, we must estimate how large a step should be taken. Once determined, we can compute the learning rate that will permit a step of that size by using Eqn. 6. This learning rate will be used to update the Λ_i's that are affected by the constraint.

The graph's current state estimate already reflects the effects of a number of other constraints. The step resulting from the addition of a new constraint should reflect the certainty of the new constraint and the certainty of the constraints already incorporated into the graph. Let gain β be the fraction of a full-step that would optimally fuse the previous estimate and the new constraint. β can be derived by differentiating the χ^2 cost of two Gaussian observations of the same quantity, or manipulated from the Kalman gain equation:

$$\beta = \Sigma_i^{-1}(\Sigma_i^{-1} + \Sigma_{graph}^{-1})^{-1} \tag{7}$$

We can estimate Σ_{graph}^{-1} from the diagonals of the information matrix: the graph's uncertainty about the transformation from pose a to b is the sum of the uncertainty of the motions between them. We have already approximated these uncertainties in our diagonal approximation to the information matrix M. In truth, the motions are correlated, but we arrive at a serviceable approximation of Σ_{graph} by summing the inverse of the diagonal elements of M between a and b. Because the Jacobians change very slowly in comparison to the state estimate, both M and these sums can be cached (rather than recomputing them every iteration). In our implementation, M (and the quantities derived from it) are updated on iterations that are powers of two.

Using Eqn. 6, we can solve for the learning rate that would result in a step of size $\sum d_i = \beta r_i$. Because there are three degrees-of-freedom per pose, we obtain three simultaneous equations for λ; we could maintain separate learning rates for each, but we use the maximum value for all three. With $\oslash$ representing row-by-row division, we write:

$$\lambda = \text{maxrow}\left(\frac{1}{b-a}\left(\beta r_i \oslash \Omega R \Sigma_i^{-1} r_i\right)\right) \tag{8}$$

This value of λ is then propagated to all of the poses after pose a:

$$\Lambda_i' = \max(\Lambda_i, \lambda) \quad \text{for } i > a \tag{9}$$

D. Processing an old constraint

When processing an old constraint, we must determine what *effective learning rate* should be used when calculating its step size. If no new constraints have ever been added to the graph, then all of the poses have identical learning rates Λ_i: the effective learning rate is just Λ_i. But if new constraints have been added, then the poses affected by the constraint might have different learning rates.

A learning rate increase caused by a new constraint can cause a large change in the state estimate, upsetting the equilibrium of other constraints in the graph. Increasing the effective learning rate of these constraints will decrease the amount of time it takes for the graph to reach a new equilibrium. If the learning rate of these older constraints was not increased, the graph would still converge to an equilibrium; however, because the learning rate could be arbitrarily small (depending on how long the optimization has been running), it could take arbitrarily long for it to do so.

A constraint between poses a and b is sensitive to changes to any of the poses between a and b: the more the poses have been perturbed (i.e., the larger the Λ_i's), the larger the effective learning rate should be. We can interpret each of the poses belonging to a constraint "voting" for the learning rate that should be used. Consequently, the effective learning rate for a constraint can be reasonably set to the average value of the learning rates between a and b. Notably, this rule has the property that it reduces to the batch case when no new constraints are added (and thus all the Λ_i's are equal).

Once the effective learning rate is computed, it can be used to compute a step according to Eqn. 5.

The effective learning rate may be greater than some of the Λ_i's of the affected poses; this must be accounted for by increasing the Λ_i's to be at least as large as the effective learning rate, as was the case when adding a new constraint.

Note that in order to avoid erroneously increasing the learning rate of poses more than necessary, any changes to the learning rate should not take effect until all constraints have been processed. For example, if there are two constraints between poses a and b, the learning rates should not be doubly increased: both constraints are responding to the same perturbation caused by a new edge.

Consider an example with three constraints: a newly-added constraint X between poses 100 and 200, and existing constraints Y (between poses 50 and 150) and Z (between poses 25 and 75). The learning rate increase caused by constraint X will cause an increase in the effective learning rate for constraint Y. On the next iteration, constraint Z will also see an increase in its effective learning rate, because constraint Y perturbed it on the previous iteration. In other words, constraint Z will be affected by constraint X, even though they have no poses in common. Their interaction is mediated by constraint Y.

This "percolating" effect is important in order to accommodate new information, even though the effect is generally small. It is, in essence, an iterative way of dealing with the correlations between constraints.

Returning to the example of buildings A and B, learning rate increases due to loop closures will propagate back toward building A in direct relationship to how tightly coupled buildings A and B are (in terms of constraints interconnecting the two). If they are coupled only by an open-loop path with no loop closures, then building A will not be affected by volatility in building B. This is because learning rates are propagated backward only via constraints involving overlapping sets of poses.

E. Algorithm Summary

In summary, adding a new constraint (or constraints) to the graph requires the following initialization:

1) Compute the constraint's effective learning rate λ using Eqn. 8 and perform a step according to Eqn. 5.
2) Increase the learning rates, as necessary:

$$\Lambda_i' = \max(\Lambda_i, \lambda) \quad \text{for } i > a \tag{10}$$

Updating an existing constraint between poses a and b involves three steps:

1) Compute the constraint's effective learning rate λ by computing the mean value of the learning rates for each

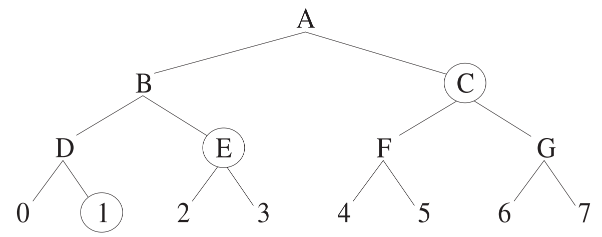


Fig. 2. Learning Rate Tree. Learning rates for each pose are stored in the leaves. Contiguous ranges of nodes can be set by modifying at most $O(\log N)$ nodes. For example, the learning rate for nodes 1-7 can be modified by adjusting three nodes: 1, E, and C. Nodes D, B, and A then must be updated as well. Similarly, cumulative sum can be implemented in $O(\log N)$ time; for example, the sum of Λ_i for $i \in [0, 5]$ can be determined by adding the sums of nodes B and F.

pose spanned by the constraint:

$$\lambda = \frac{1}{b-a} \sum_{a+1}^{b} \Lambda_i \tag{11}$$

2) Compute and apply the constraint step, using learning rate λ;
3) Update the learning rates of the affected poses, as in Eqn. 10.

After processing all of the constraints, the learning rates Λ_i are decreased according to a generalized harmonic progression, e.g.:

$$\Lambda_i' = \frac{\Lambda_i}{1 + \Lambda_i} \tag{12}$$

Note that these rules guarantee the increasing monotonicity of the Λ_i at any given time step. In other words, $\Lambda_i \leq \Lambda_j$ if $i < j$. While any particular Λ_i tends to decrease over time, it does not necessarily decrease monotonically due to the learning rate increases caused by new constraints.

F. Learning Rate Data Structure

An obvious data structure for maintaining the values Λ_i is an array. The operations required by Eqn. 10 and Eqn. 11 would run in $O(N)$ time for a graph with N poses. This would be worse than the batch algorithm, which has $O(\log N)$ complexity per constraint.

Fortunately, an augmented balanced binary tree can be used to implement both of these operations in $O(\log N)$ time. Each pose is represented by a leaf node. Other nodes maintain the minimum, maximum, and sum of their children, with the special caveat that in the case when the minimum and maximum values are the same, the child nodes are overridden. It is this "overriding" behavior which allows the write operations to be performed in $O(\log N)$ time.

For example, implementing Eqn. 10 will affect only a contiguous set of indices starting at some index $j \geq i$ (see Fig. 2). Every member of this set can be overridden by modifying no more than $O(\log N)$ ancestor nodes in the tree. The ancestors' parents also need to be visited so that their min/max/sum fields can be updated, but there are at most $O(\log N)$ parents that require updating.

Eqn. 11 is most easily implemented using a primitive operation that computes the cumulative sum $\sum_{j=0}^{i} \Lambda_j$; this is done by adding together the $O(\log N)$ sums that contribute to the cumulative sum (taking care to handle those nodes that override their children). The mean over an interval is then the difference of two cumulative sums divided by the number of indices in the interval.

The implementation of this data structure is relatively straightforward, if tedious. We refer you to our source code for the implementation details, at http://rvsn.csail.mit.edu.

G. Analysis

On a graph with N poses and M constraints, memory usage of the algorithm is $O(N + M)$, the same as the batch algorithm. Runtime per constraint also remains $O(\log N)$, though constraints are usually processed in "full iterations", in which all the constraints are processed in a rapid succession.

The actual CPU requirements in order to obtain a "good" result are difficult to specify. Our approach does not guarantee how much the error will decrease at each step; the error can even increase. Consequently, we cannot provide a bound on how many iterations are required for a particular level of performance.

That said, the convergence of the algorithm is typically quite rapid, especially in correcting gross errors. Quality requirements naturally vary by application, and iterative approaches, like the one presented here, offer flexibility in trading quality versus CPU time.

The classical stochastic gradient descent algorithm picks the constraints at random, however, we typically process the constraints in a fixed order. The additional randomization caused by processing the constraints in different orders may have a small positive effect on convergence rate. Processing the constraints in a fixed order, however, causes the graph to vary more smoothly after each full iteration.

IV. Constraint Scheduling

In the case of the batch algorithm, each *full iteration* includes an update step for *every* constraint in the graph. The learning rate is controlled globally, and the graph converges more-or-less uniformly throughout the graph.

In the incremental algorithm that we have described, the learning rate is not global, and different parts of the graph can be in dramatically different states of convergence. In fact, it is often the case that older parts of the graph have much lower learning rates (and are closer to the minimum-error configuration) than newer parts, which are farther from a minimum.

This section describes how the least-converged parts of the graph can be optimized, without the need to further optimize distant and already well-converged parts.

The least-converged part of the graph is typically the most important because they generally contain the robot itself. The area around the robot is usually the least-converged part of the graph because new constraints are added to the graph based on the observations of the robot. Consequently, it is critically important for the robot to be able to improve its local map, and it is relatively unimportant to "fine tune" some distant (and often not immediately relevant) area.

Our method does not require a map to be explicitly segmented (into buildings, for example): rather, we automatically identify the subgraph that is the most volatile (i.e., has the largest learning rates), then determine the set of constraints that must be considered in order to reduce the learning rates of that subgraph. This subset of constraints is typically much smaller than the total set of constraints in the graph, resulting in significant CPU savings.

Here's the basic idea: suppose we want to reduce the maximum Λ_i to Λ'_{max} during an iteration. All of the constraints in the graph have an effective learning rate either larger or smaller than Λ'_{max}. Those constraints with larger effective learning rates must be processed before the Λ_i's are reduced, because those constraints still need to take larger steps.

In contrast, those constraints that have smaller effective learning rates are taking comparatively small steps: their small steps tend to be ineffective because other constraints are taking larger steps. Processing constraints with small effective learning rates will generally not achieve any χ^2 reduction when there are other constraints taking large steps; we can save CPU time by skipping them.

The following algorithm implements this heuristic to optimize only a recent subgraph of the pose graph:

1) Look up the maximum learning rate in the graph, e.g., $\Lambda_{max} = \Lambda_{nposes-1}$. If we performed a full iteration, the maximum learning rate after the iteration would be $\Lambda'_{max} = \Lambda_{max}/(1 + \Lambda_{max})$. We use this as our target value.
2) Perform update steps only on those constraints whose effective learning rate is greater than Λ_{max}.
3) Set $\Lambda'_i = \Lambda'_{max}$ for all $i \geq p$.

In other words, this algorithm reduces the *maximum* learning rate in the graph by a full harmonic progression by computing and operating on only a subset of the graph's constraints. The procedure conservatively identifies the set of constraints that should be considered. Note that the operation in step 3 can also be implemented in $O(\log N)$ time using the previously-described learning rate data structure.

In many cases, large reductions in learning rate can be achieved by considering only a handful of constraints. In the Freiburg data set, the technique is very effective, with under 10% of the constraints updated at each iteration (see Fig. 3). Since computational cost is linear in the number of constraints processed, this yields a speed-up of almost 10x. Despite the fact that a only a small fraction of the constraints are considered, the χ^2 error is essentially the same as the much slower implementation that considers all constraints (see Fig. 4). This is because the bulk of error in the graph is concentrated in the more recent portions of the graph. The "all constraints" method spends large amounts of CPU time tweaking distant and already well-converged portions of the graph, which generally does not yield very large χ^2 reductions.

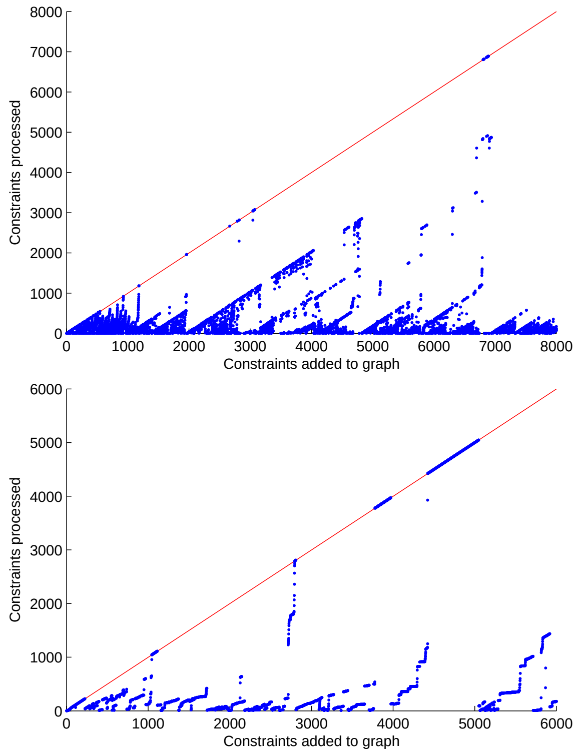


Fig. 3. Constraint Scheduling. Top: Freiburg, Bottom: Intel Research Center. Selective processing of constraints leads to large speed-ups. In the figure, the line indicates the total number of constraints in the graph; the points show the actual number of constraints that were processed. In the Intel dataset, more constraints tend to be processed since the robot repeatedly revisits the same areas, creating new constraints that span many poses, thus disturbing larger portions of the graph.

In contrast, the "selected constraints" method focuses solely on the parts of the graph that will lead to the largest χ^2 reductions.

The same approach on the Intel Research Center data set processes 27% of the constraints on average. The lower performance is due to the fact that the robot is orbiting the entire facility, frequently creating new constraints between poses that are temporally distant. This causes learning rate increases to propagate to more poses. Despite this, a significant speedup is achieved.

The approach outlined here is somewhat greedy: it attempts to reduce the worst-case learning rate to Λ'_{max}. It is Λ'_{max} that determines the number of constraints that are necessary to perform the partial update. It is possible that a slightly larger value of Λ_{max} would result in significantly fewer constraints to process, resulting in larger χ^2 reduction per CPU time. This is an area of future work.

V. Results

We compared the runtime performance characteristics of our approach to that of LU Decomposition (non-linear least squares), the Extended Kalman Filter (EKF) and Gauss-Seidel Relaxation (GS). Our testing was performed on a modern desktop system with a 2.4GHz CPU, and our code was written in Java.

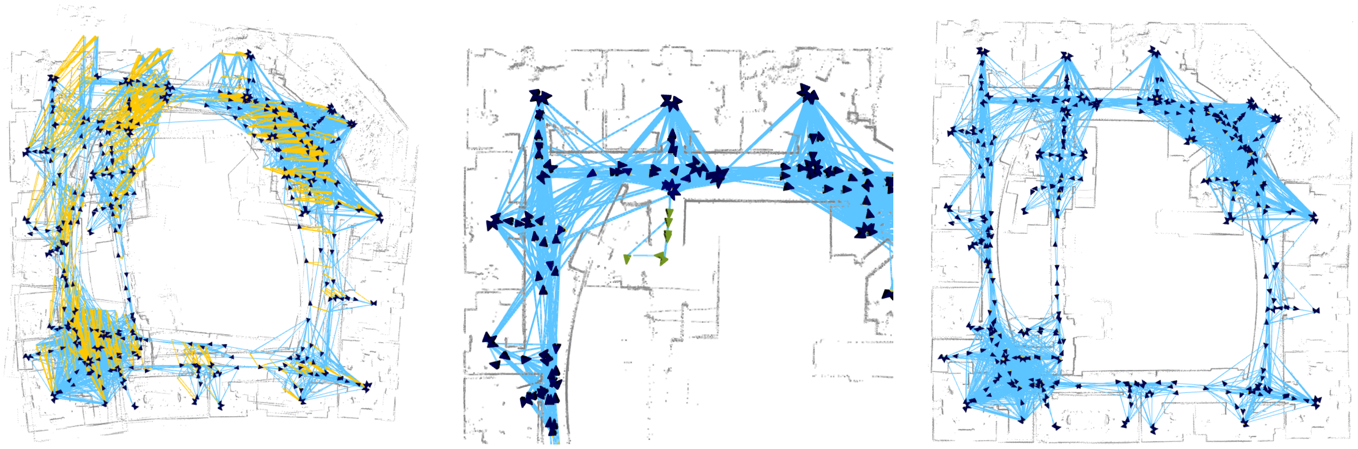
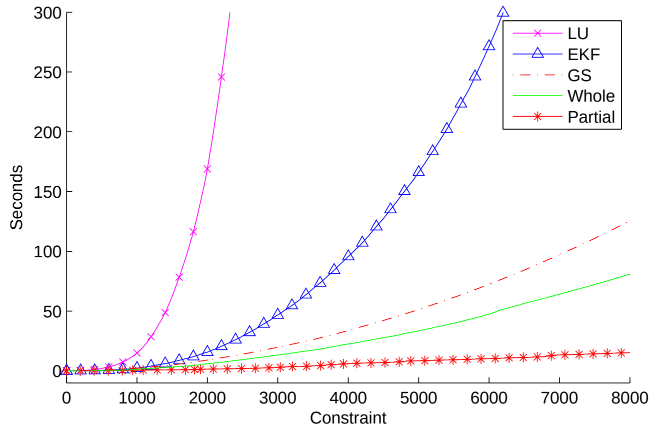


Fig. 6. Intel Research Center. Left: the open-loop trajectory. Middle: After orbiting the facility three times, the robot is entering a new area; the well-explored area has a low learning rate while the newly entered area has a high learning rate. Right: the posterior map.



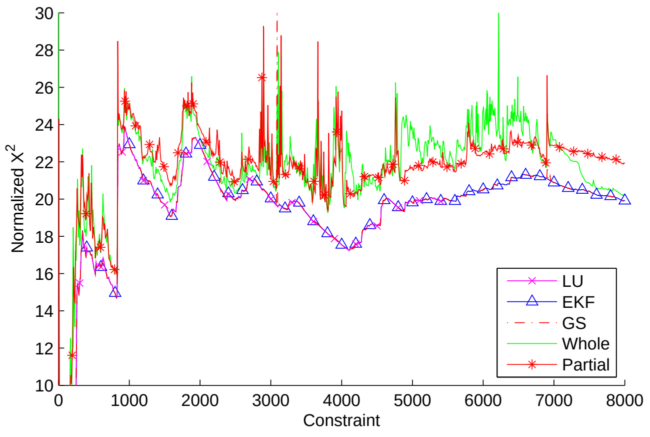


Fig. 4. Cumulative Runtime and Error Comparisons, Freiburg dataset. Each constraint was added one at a time. EKF and LU computational time dwarfs the others. Our proposed method (with constraint selection) is by far the fastest at 21 seconds; our method (without constraint selection) beats out Gauss-Seidel relaxation. In terms of quality, LU, EKF, and Gauss-Seidel all produce nearly optimal results; our methods have marginally higher χ^2 error, as expected, but the maps are subjectively difficult to distinguish. (Partial: 15.5s, Whole: 82.5s, Gauss Seidel: 128.3s, EKF: 650s.)

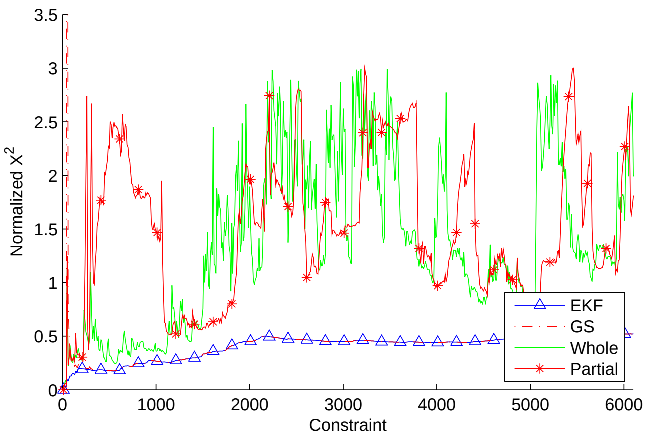


Fig. 5. Error Comparisons, Intel Research Center. Our methods are able to stay relatively close to the nearly-optimal χ^2 error produced by the EKF and Gauss-Seidel, however, did so at a fraction of the run time. (Partial: 14.5s, Whole: 45.4s, Gauss-Seidel: 65.2s, EKF: 132.3s)

Since this algorithm is targeted at on-line applications, we assume that the robot requires a full state estimate after every observation; this makes the performance of the EKF no worse than that of an information-form filter which would require frequent inversions of the information matrix. The CPU time and χ^2 results on the Freiburg data set are shown in Fig. 4. Similar behavior occurred on the Intel Research Center dataset (see Fig. 5).

To mitigate any potential advantage of the iterative algorithms (they could, after all, be extremely fast by simply doing no work), they were forced to continue to iterate until the χ^2 was reduced below a threshold (Freiburg 25, Intel 3.0).

Our approach, especially with constraint selection enabled, is significantly faster than any of the other methods. In terms of quality (as measured by χ^2 error), our approaches produced somewhat worse maps. However, the difference is very subtle. When using the constraint selection algorithm (figures labeled "partial"), our algorithm is significantly faster than the other approaches.

Even if all constraints must be processed, the algorithm is very fast. After adding the last constraint, processing all 8000 constraints on the Freiburg graph with 906 poses required 16.8ms. When using the constraint selection algorithm, only a small fraction of these constraints need to be processed: the algorithm took an average of 1.1ms to add each of the final 10 constraints on the Freiburg graph: for each, it considered an average of 73 constraints. Several Freiburg maps are illustrated in Fig. 1, including the learning rates (as a function of pose).

Putting these numbers in perspective, the Intel Research Center data set represents 45 minutes of data; incorporating observations one at a time (and outputting a posterior map after every observation) required a total cumulative time of 14.5s with constraint selection enabled, and 45.4s without. This would consume about 0.6% of the robot's CPU over the lifetime of the mission, making the CPU available for other purposes. These maps were of fairly high quality, with χ^2 errors only marginally larger than that of the EKF.

Several maps from the Intel Research Center are shown in Fig. 6. The open-loop trajectory is shown, as well as several intermediate maps. In the first map, a loop closure is just about to occur; prior to this, the learning rate is low everywhere. After the loop closure, the learning rate is high everywhere. The final map exhibits sharp walls and virtually no feature doubling; the incremental algorithm matches the quality of the batch algorithm.

As with the batch algorithm, the optimization rapidly finds a solution *near* the global minimum, but once near the minimum, other algorithms can "fine tune" more efficiently. On a large synthetic data set, our method requires 24.18s for incremental processing; after a post-processing using 2.0s of Gauss-Seidel relaxation, the normalized χ^2 is 1.18. This is far better than Gauss-Seidel can do on its own: it requires 168s to achieve the same χ^2 on its own.

VI. Conclusion

We have presented an incremental non-linear SLAM algorithm, generalizing an existing batch algorithm. Our introduction of a spatially-dependent learning rate improves CPU efficiency by limiting learning rate increases to only those areas of the map that require them. We also showed how to optimize only the subgraph that has the largest learning rate, which leads to significant performance improvements.

Iterative non-linear methods, like the one presented here, offer many advantages over conventional SLAM algorithms including faster operation, lower memory consumption, and the ability to dynamically trade CPU utilization for map quality.

References

[1] 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.

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

[3] A. Gelb, *Applied Optimal Estimation.* Cambridge, MA: MIT Press, 1974.

[4] S. Julier and J. Uhlmann, "A new extension of the Kalman filter to nonlinear systems," in *Int. Symp. Aerospace/Defense Sensing, Simul. and Controls, Orlando, FL*, 1997, pp. 182–193. [Online]. Available: citeseer.ist.psu.edu/julier97new.html

[5] 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.

[6] F. Lu and E. Milios, "Globally consistent range scan alignment for environment mapping," *Autonomous Robots*, vol. 4, no. 4, pp. 333–349, 1997.

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

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

[9] F. Dellaert, "Square root SAM," in *Proceedings of Robotics: Science and Systems*, Cambridge, USA, June 2005.

[10] 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.

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

[12] E. Olson, J. Leonard, and S. Teller, "Fast iterative optimization of pose graphs with poor initial estimates," in *Proceedings of ICRA 2006*, 2006, pp. 2262–2269.

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

[14] F. Lu and E. Milios, "Robot pose estimation in unknown environments by matching 2d range scans," in *CVPR94*, 1994, pp. 935–938. [Online]. Available: citeseer.ist.psu.edu/lu94robot.html

[15] S. Se, D. Lowe, and J. Little, "Vision-based mobile robot localization and mapping using scale-invariant features," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Seoul, Korea, May 2001, pp. 2051–2058. [Online]. Available: citeseer.ist.psu.edu/se01visionbased.html