Hybrid Hessians for Flexible Optimization of Pose Graphs

Matthew Koichi Grimes
Dragomir Anguelov
Yann LeCun

Abstract—We present a novel “hybrid Hessian” six-degrees-of-freedom simultaneous localization and mapping (SLAM) algorithm. Our method allows for the smooth trade-off of accuracy for efficiency and for the incorporation of GPS measurements during real-time operation, thereby offering significant advantages over other SLAM solvers.

Like other stochastic SLAM methods, such as SGD and TORO, our technique is robust to local minima and eliminates the need for costly relinearizations of the map. Unlike other stochastic methods, but similar to exact solvers, such as iSAM, our technique is able to process position-only constraints, such as GPS measurements, without introducing systematic distortions in the map.

We present results from the Google Street View database, and compare our method with results from TORO. We show that our solver is able to achieve higher accuracy while operating within real-time bounds. In addition, as far as we are aware, this is the first stochastic SLAM solver capable of processing GPS constraints in real-time.

I. INTRODUCTION

In mobile robotics, the ability to self-localize is a critical prerequisite to many higher-level functions. SLAM can be described as the iterative process of localizing oneself relative to previously explored locations, and mapping new locations relative to oneself.

A popular formulation of this problem is to model it as a graph of nodes, representing robot poses to be solved for, and edges, representing sensor readings. Each edge defines an energy function of the nodes it connects. This function penalizes deviations between the nodes’ relative pose, and the relative pose as measured by a sensor. By minimizing the sum of these constraint energies with respect to the pose variables, we solve for the maximum a posteriori values of all poses given all sensor readings.

In this paper, we draw inspiration from two divergent approaches to minimizing this energy: full linearized solvers, and stochastic relaxation. Full solvers are mathematically equivalent to the well-known Gauss-Newton method, though with better numerical stability and much-improved speed. These methods linearize the total constraint energy $E$ around the current value of the poses $z$, then solve the resulting linear equation for an update $dz$ that minimizes $E(z + dz)$ in a least-squares sense. Sparse solving techniques can incrementally solve for this update in real time, but once the update becomes too large, the map must be relinearized from a new linearization point and solved from scratch, an expensive process. Furthermore, like Gauss-Newton, these methods are susceptible to becoming stuck in local minima.

By contrast, stochastic relaxation relaxes one constraint at a time without need of a fixed linearization point. Like the stochastic gradient descent techniques popular in machine learning, stochastic relaxation converges more quickly than performing full Newton steps, and is more robust to local minima. To minimize oscillation between competing constraints, stochastic relaxation encodes nodes in a hierarchical pose tree, where each pose is defined relative to its parent (see fig. 1). For each constraint, there is a difference $r$ between a pose and a constraint’s desired value for that pose. Stochastic relaxation minimizes this difference by accumulating $-\alpha_i r$ into the node’s ancestors $n_i$ in the tree (the coefficients $\alpha_i$ are chosen to sum to one), which has the effect of shifting the node by $-r$. While efficient, this can introduce a systematic distortion into the map, known as the “dog-leg problem” [1]. As shown in fig. 2, this arises when a constraint has a large error in position but little error in rotation, which may happen by chance, or because the constraint only acts on position, as is the case with GPS.

In this paper, we present a method that performs stochastic relaxation, but which does so by solving a set of linear equations. This eliminates the dog-leg problem. It also allows the user to smoothly transition between stochastic and exact updates at run-time, flexibly trading off cost for accuracy as needed.

II. RELATED WORK

Early SLAM algorithms, such as Smith and Cheeseman’s EKF SLAM [2], incorporated new observations into an extended Kalman filter (EKF), which tracked the pose of the robot and any landmarks. The approach is used to this day, such as with Davison’s monocular visual SLAM [3], and Kim and Sukkarihe’s GPS-augmented SLAM for flying vehicles [4]. Unfortunately, EKF-based SLAM requires a full inversion of a dense N-by-N matrix with each new observation, where N is the size of the state vector containing all tracked landmark and robot poses. This $O(N^3)$ operation scales poorly with the size of the map. It can be mitigated by selective sparsification of the information matrix [5], [6], but reconstructing the map from this matrix remains costly.

The “full SLAM problem” solves for the entire history of robot poses, not just the most recent. This paper concerns itself with the pose graph formulation of this problem. As described in section I, two recent approaches to this have been full linear solvers, as in $\sqrt{SAM}$ [7] and iSAM [8], and stochastic relaxation, as used by SGD [9] and TORO [10]. The $\sqrt{SAM}$ method and its descendants speed up this linear solve through column reordering [7] and incremental factorization [8], arriving at the same exact answer as the Gauss-Newton method in much shorter time. Drawbacks include the computational cost of relinearizations, and susceptibility to local minima.

SGD [9] performs stochastic relaxation in two dimensions by parametrizing each pose in the pose hierarchy as an addition onto the pose of its parent. This allows a constraint to be relaxed efficiently by simply adding fractions of its residual into the constrained pose’s ancestors. TORO [10] extends this into three dimensions by updating translation and rotation separately to work around the non-commutativity of rotations in 3D. Both methods are highly robust to large residuals and

M. Grimes and Y. LeCun are with the Courant Institute for Mathematical Sciences, New York University, 719 Broadway, New York NY 10003, USA. mkgr@cs.nyu.edu. D. Anguelov is with Google Inc, 1600 Amphitheatre Parkway, Mountain View, CA drago@google.com
and rotates its entire sub-tree, leaving the relative pose from the domain, since translating or rotating the root translates the path the constraint’s “domain”. The root is excluded from the constraint’s “root”, and all other nodes in the tree excluding the node with the smallest tree depth. We call this set of poses whose values affect the constraint energy. The constraint’s root is the topmost node in the path from one node of the constraint to the other. It is not part of the constraint domain.

The dog-leg problem occurs when an error in position is corrected by position updates only, without also updating rotations. Here we see a relaxation of the red constraint, with and without the dog-leg problem.

local minima, but both suffer from dog-leg distortions as described in section I.

Some authors have proposed solving for only those poses in the pose graph that are near the current pose. This can help SLAM adapt to dynamic environments [11], or save on-board cost by leaving the task of creating a globally consistent map to an off-board computer [12]. Our work can be easily adapted to solve for local poses only (see section III-E and III-F)). However, storing the map in a single global coordinate frame can have many advantages to real-time tasks. For example, it can facilitate detecting loop-closures, by allowing the robot to only compare the current frame against those thought to be nearby in the global coordinate frame. A global coordinate system can likewise help align multiple maps in collaborative mapping, or help register a map in progress against satellite imagery or other geolocated data. For these reasons, we have chosen to focus on real-time, globally consistent SLAM.

III. ALGORITHM

In this section, we first describe our parametrization, introducing relevant terminology. We review the mathematics of performing a full linearized update to the poses, then introduce our stochastic approach in terms of this formalism. We describe methods of reducing the cost of expensive updates, and of solving more than one constraint at a time, for stable processing of GPS constraints. We then present the algorithm in summary.

A. Pose Trees

Like [10], we represent our poses by initially growing a spanning tree out of a node in the pose graph, and defining each pose relative to its parent in the tree. Fig. 1 shows a pose graph, and one possible tree ordering. In this parametrization, the energy of a constraint connecting nodes a and b can only be affected by nodes in the path through the tree from a to b, excluding the node with the smallest tree depth. We call this the “domain” of the constraint’s “root”, and all other nodes in the path the constraint’s “domain”. The root is excluded from the domain, since translating or rotating the root translates and rotates its entire sub-tree, leaving the relative pose from a to b unchanged. Likewise, minimizing a constraint’s residual only changes nodes in its domain. The node from which this tree is grown becomes the root of the entire tree. We allow for the use of GPS and other constraints that operate in “absolute” coordinates, by using a root node that represents the earth’s frame. GPS readings can then be represented as relative position constraints between the earth and a pose. Because the earth node is the root of the tree, it is outside of any constraint’s domain, and is thus appropriately left unchanged by the pose graph optimization.

We initialize the poses using the tree constraints, by concatenating their desired relative poses down from the root node. Using noisy constraints in the tree, such as GPS constraints, can lead to poor initial poses. For this reason, it is important to prefer high-stiffness edges (corresponding to low-noise sensors) when building the spanning tree. An MST algorithm that maximizes the total stiffness of tree edges is one option. The results in this paper come from a simple breadth-first traversal, modified to avoid GPS edges whenever possible.

B. Linearized updates

We represent our poses as 7-dimensional vectors composed of a position vector and quaternion (quaternions are re-normalized after each update). We collect all poses in a single vector z. For a constraint c connecting poses a and b. Let be a function of the relative pose between a and b. We define the constraint energy as a quadratic penalty on the difference between and its desired value , weighted by distance matrix :

\[ E_c(z) = (f_c(z) - k_c)^T S_c (f_c(z) - k_c) \]

(1)

For example, (1) may represent a relative position constraint by having be the relative position of b in the frame of a, and having be this relative position as measured by a sensor, such as a wheel encoder. Likewise, (1) may model a relative rotation constraint by setting and , where is the relative rotation of b in the frame of a, is the relative rotation as measured by a sensor, and is the identity rotation. In EKF terminology, is the prediction function, is the “observation”, and is the inverse of the sensor covariance matrix.

The total energy is the sum of constraint energies:

\[ E = \sum_c E_c(z) \]

(2)

We linearize around , the current value of z, using the substitution \( z = \bar{z} + x \), where x is the parameter update we
We plug (3) and (4) into (1) and (2) to get:

\[ J \approx f_c(\bar{z}) + \frac{\partial f}{\partial z} |_{\bar{z}} x \]  

(3)

\[ S_c = L_cL_c^T \]  

(4)

We plug (3) and (4) into (1) and (2) to get:

\[ E = \sum_c \left\| L_c^T (f_c(\bar{z}) + \frac{\partial f}{\partial z} |_{\bar{z}} x - k_c) \right\|^2 \]  

(5)

Let \( J_c = L_c^T \frac{\partial f}{\partial z} |_{\bar{z}} \) and \( r_c = L_c^T (k_c - f_c(\bar{z})) \) to get:

\[ E = \sum_c \left\| J_c x - r_c \right\|^2 \]  

(6)

\[ = \left\| J x - r \right\|^2 \]  

(7)

Here matrix \( J \) and column vector \( r \) are simply the individual constraints’ \( J_c \) and \( r_c \) stacked vertically:

\[ J = \begin{bmatrix} J_{c1}^T & \cdots & J_{cM}^T \end{bmatrix}^T, \quad r = \begin{bmatrix} r_{c1}^T & \cdots & r_{cM}^T \end{bmatrix}^T \]  

(8)

We arrive at a linear least-squares problem

\[ x = \min_x \left\| J x - r \right\|^2, \]  

(9)

which can be solved using one of two standard methods: normal equations and orthogonal decomposition. The normal equations are obtained by taking the derivative of \( E \) in (7) and setting it equal to zero:

\[ 2J^T (J x - r) = 0 \]  

(10)

\[ H x = J^T r, \]  

(11)

where \( H \) denotes \( J^T J \), as it is the approximated Hessian from the Gauss-Newton method. Equation 11 may be upper-triangularized by the Cholesky decomposition \( H = R^T R \), followed by one back-substitution:

\[ R^T R x = J^T r \]  

(12)

\[ R x = b \]  

(back-substituted)  

(13)

With another back-substitution, we solve for \( x \). Using orthogonal decomposition, we also arrive at 13 by setting \( J x - r = 0 \) and QR-factorizing \( J \):

\[ J x = r \]  

(14)

\[ Q R x = r \]  

(15)

\[ R x = Q^T r \]  

(16)

\[ R x = b \]  

(17)

When \( J \) is nearly upper-triangular, orthogonal decomposition can take \( O(N^2) \) time to solve compared to \( O(N^3) \) for the normal equations, a fact we exploit in section III-D to efficiently solve for stochastic updates. Having solved for update \( x \), we add it to pose parameters \( z \):

\[ z \leftarrow z + x \]  

(18)

This is followed by normalizing the quaternions in \( z \).

---

**Algorithm 1 OptimizePoseGraph**

Growth a spanning pose tree through the pose graph. Sort constraints \( C \) in increasing order of root depth. \( G \leftarrow \{\} \) ▶ empty set

for \( c \) in \( C \) do

if \( ||G|| > \text{gps} \cdot \text{batch size} \) then

BatchUpdate(\( G, D_{\text{max}} \))

\( G \leftarrow \{\} \)

end if

if \( c \) is a GPS constraint then

Add \( c \) to \( G \)

else

BatchUpdate(\( G, D_{\text{max}} \)) ▶ no-op if \( G \) is empty

Update(\( c, D_{\text{max}} \))

end if

end for

---

C. Stochastic Updates

The update \( x \) described above is expensive to evaluate, since it is calculated using all constraints in the graph. An alternative is to inexpensively calculate one approximate update \( x_c \) for each constraint \( c \). Such “stochastic” updates have a long history in machine learning[13], as they can converge much more quickly than exact updates, and provide some robustness to local minima. Oscillations due to the inexactness of these updates are mitigated by using a decaying learning rate, as will be discussed in section III-H. In this section we derive our expression for \( x_c \).

Plugging (8) into the right-hand-side of (11) gets:

\[ H x = \sum_c J_c^T r_c \]  

(19)

Note that we may express the total update \( x \) as a sum \( x = \sum_c x_c \) of constraint-specific updates \( x_c \), where:

\[ x_c = H^{-1} J_c^T r_c \]  

(20)

In practice, we compute \( x_c \) by solving the linear system:

\[ H x_c = J_c^T r_c \]  

(21)

In stochastic relaxation, the parameters \( z \) are updated by one \( x_c \) at a time, rather than by their sum, \( x \). In many applications, this converges quicker, and typically computation is saved by not recalculating \( H = \sum_c J_c^T J_c \) from scratch after each constraint update.

Our goal is to speed up solving for \( x_c \) and make it real-time. One can use an approach similar to second-order back-propagation in neural networks [14], where \( H \) is reduced to a diagonal by zeroing all off-diagonal elements. Unfortunately, while this makes solving for \( x_c \) linear in the number of non-zeros in vector \( J_c^T r_c \), it can prevent convergence. The reason is that this can greatly reduce the matrix norm \( ||H|| \), making the resulting update \( x_c \) very large. Large updates to the poses, particularly to the rotations, can easily prevent convergence, since rotating a node rotates its entire sub-tree. In [9], Olson prevents this by using an approximation to \( J_c \) where each non-zero block is replaced by a constant diagonal matrix. This eliminates any large derivatives it may contain, and removes the need to calculate any derivatives. While the resulting algorithm is very fast, it erases the correlation between
rotation parameters and position residuals, causing the “dog-leg problem”. TORO [10] employs a similar simplification in 3D, with the same problem.

D. Hybrid Hessians

We now describe our approximation to $H$, which is easy to invert, avoids the dog-leg problem, and does not produce overly large updates. We approximate $H$ in (21) by a “hybrid Hessian” $H_c$ specific to constraint $c$. Note that $H$ is composed of $N$ by $N$ blocks, where $N$ is the number of poses. We define $H_c$, as the full Hessian $H = \sum_c (J^c_T J_c)$ with all of the off-diagonal blocks zeroed except those of $J^c_T \hat{J}_c$, namely

$$H_c = J^T c J_c + B_c$$

where $B_c$ is the block-diagonal approximation to the hessian built from all constraints except for constraint $c$:

$$B_c = \mathcal{B}(H) - \mathcal{B}(J^T c J_c)$$

Here $\mathcal{B}$ is an operator which zeros any off-diagonal blocks. In practice, these blocks are never calculated in the first place, so that $\mathcal{B}(J^T c J_c)$ is calculated in $O(d)$ time, where $d$ is the number of nonzero blocks in $J_c$. Instead of recalculating $\mathcal{B}(H) = \sum_c \mathcal{B}(J^T c J_c)$ at each constraint relaxation, we update the approximation in $O(d)$ time by:

$$\mathcal{B}(H) \rightarrow \mathcal{B}(H) - \mathcal{B}(J^T c J_c)_{old} + \mathcal{B}(J^T c J_c)$$

where $\mathcal{B}(J^T c J_c)_{old}$ denotes the value of $\mathcal{B}(J^T c J_c)$ calculated in the previous relaxation of constraint $c$.

We obtain our stochastic update $x_c$ by replacing $H$ in (21) with $H_c$, and solving for $x_c$:

$$(J^T c J_c + B_c)x_c = J^T c r_c$$

This yields a solution to the following least-squares problem:

$$\min_{x_c} \left( \|J_c x_c - r_c\|^2 + \|\Gamma_c x_c\|^2 \right)$$

where $\Gamma_c$ is the upper triangle of the Cholesky factorization:

$$B_c = \Gamma_c^T \Gamma_c$$

Without $\|\Gamma_c x_c\|^2$, (26) would be an under-constrained minimization problem for a single constraint $c$. We have regularized it using the block-diagonals of the full hessian, both to make it solvable, and also to prevent each constraint update from simply satisfying constraint $c$ without regard to all other constraints.

Only the poses in constraint $c$’s domain $D_c$ affect $c$’s energy, as seen in fig. 1. We can therefore solve a reduced-dimension version of (25) by omitting all rows and columns that do not correspond to poses in $D_c$. We denote this omission using hats ($\hat{\cdot}$), as in:

$$(\hat{J}^T c \hat{J}_c + \hat{B}_c)\hat{x}_c = \hat{J}^T c r_c$$

One could solve this dense normal equation using Cholesky factorization. But since this is cubic in the size of $D_c$, it may be unacceptably expensive for real-time performance, because constraints near the bottom of large pose trees can have large constraint paths. Instead, we solve (26) using the following orthogonal decomposition:

$$\begin{bmatrix} \hat{J}_c \\ \Gamma_c \end{bmatrix} \hat{x}_c = \begin{bmatrix} r_c \\ 0 \end{bmatrix}$$

Note that left-multiplying both sides of (29) by $[ \hat{J}^T c \hat{J}_c \hat{\Gamma}^T c ]$ recovers (28). Equation 29 is a Tikhonov regularization of the under-constrained problem $J_c \hat{x}_c - \hat{r}_c = 0$, with the Tikhonov matrix $\hat{\Gamma}_c$ constructed from diagonal blocks of the hessian $H$.
Denoting the i'th block of $\hat{J}_c$ as $J^i_c$ and the i'th block of the block-diagonal matrix $\Gamma_c$ as $\Gamma^i_c$ we rewrite (29) as

$$
\begin{bmatrix}
J^0_c & J^1_c & J^2_c & \ldots & J^{d-1}_c \\
\Gamma^0_c & \Gamma^1_c & & & \\
\Gamma^1_c & \Gamma^2_c & \ldots & \\
& \ddots & \ddots & \\
& & \Gamma^{d-1}_c & \\
\end{bmatrix}
\begin{bmatrix}
x_c \\
0 \\
0 \\
\vdots \\
0
\end{bmatrix}
= \begin{bmatrix}
r_c \\
0 \\
0 \\
\vdots \\
0
\end{bmatrix}.
$$

Here, $d$ is the size of domain $D_c$. Each block is 7 by 7, since 7 is the number of dimensions in a single pose or residual. Equation 30 is therefore nearly upper-triangular. This allows us to fully upper-triangularize it (using Givens rotations) in $O(d^3)$ time, not the usual $O(d^5)$ for dense matrices. The subsequent back-substitution to solve for $x$ takes $O(d^2)$ time as well. Updating $B_c$ is $O(d)$. The total cost of relaxing a constraint is therefore $O(d^2)$. For a relatively balanced tree, we can estimate the expected path length for a constraint as $d_e = O(\log(N))$, where $N$ is the total number of poses. This is because the domain size $d$ of a constraint $c$ is one less than the length of the tree path connecting the two nodes constrained by $c$. The expected running time for a single iteration through all $M$ constraints is therefore $O(Md_e^2)$, or $O(M\log(N)^2)$.

E. Interpolated solving

In large pose trees with low branching factor (such as urban pose trees), the path length for some constraints can get into the thousands, making even $O(d^2)$ too costly for real-time operation on a single processor. Fortunately, it is possible to solve for an approximation to $x_c$ within a user-chosen computational cost budget which can range from $O(d)$ to $O(d^2)$. This is done by solving for a subset $S_c$ of the nodes in the path $D_c$, then distributing these updates over the remaining nodes.

1) Merging constraints: in (28), we solved for only the nodes in $D_c$ by omitting from (25) the rows and columns corresponding to other nodes. We take the same approach here, solving for only the nodes in $S_c \in D_c$ by omitting rows and columns in (28). When omitting nodes from the path, we are replacing chains of constraints between unomitted nodes with single constraints. This change affects $B_c = \sum_{i \neq c} B(J^i_c J_i)$. Consider a pair of nodes $a$ and $b$ in $S_c$, where $a$ is $b$'s closest ancestor in $S_c$, or if none exists, the constraint root (fig. 4). We replace a chain of constraints between $a$ and $b$ by a single constraint $\beta$. Let $\alpha$ be $b$'s current parent constraint in the path. We update $B$ as follows:

$$
B \leftarrow B - J^T_\alpha J_\alpha + J^T_\beta J_\beta
$$

Both $\alpha$ and its replacement $\beta$ connect two consecutive nodes in the path. As can be seen in fig. 1, such constraints have only one node in their domain, namely the lower of the two nodes they connect (in this case, $b$). Both $J_\alpha$ and $J_\beta$ therefore have only one nonzero block, making the update in (31) an $O(1)$ operation. The path from $a$ to $b$ may contain another constraint $\gamma$, but we need not subtract $J^T_\gamma J_\gamma$ from $B$, since its domain node has been omitted from the path, and its corresponding rows and columns are not included in the linear solve. Note that modifying $B$ for all nodes in $S_c$ is linear in the size of $S_c$, since we perform the update in (31) for each node in $S_c$ whose parent was omitted from $S_c$.

To calculate the $J_\beta$ of merged constraint $\beta$, we need its stiffness $S_\beta$ and desired value $k_\beta$, where $k_\beta$ follows directly from $\beta$'s desired relative pose $p_\beta$ (see (1)). If $c_1 \ldots c_n$ are the constraints merged to create $\beta$, we get $p_\beta$ by taking the product of the desired transforms of $c_1 \ldots c_n$:

$$
p_\beta = \prod_{i=1}^{n} p_i
$$

Since stiffness is the inverse of sensor covariance, we find the merged stiffness $S_\beta$ by applying the covariance merging rule $C_{merged} = (\sum_i C_i^{-1})^{-1}$. In terms of stiffnesses this becomes:

$$
S_\beta = \sum_{i=1}^{n} R_{ai} S_i R^T_{ai}
$$

where $S_i$ is the stiffness of $c_i$ and $R_{ai}$ is the desired rotation from node $a$ to $i$.

2) Solving and distributing the update: After modifying $B$ with (31) and eliminating the omitted nodes' rows and
columns from (29), we get the reduced orthogonal decomposition:

\[
\begin{bmatrix}
\hat{J}_c \\
\Gamma_c
\end{bmatrix} \tilde{x}_c =
\begin{bmatrix}
\mathbf{r}_c \\
\mathbf{0}
\end{bmatrix}
\] (34)

Here, \( \hat{J}_c \) is created from the Cholesky decomposition \( \Gamma_c^T \Gamma_c = B \), where \( B \) is updated by (31), retaining only the rows and columns corresponding to nodes in \( S_c \). After solving for \( \tilde{x}_c \), we revert the modified blocks of \( B \) to their previous values before updating by 24. If we apply update \( \tilde{x}_c \) to \( z \) as before, the path can potentially bend only at the nodes in \( S_c \), making the chain discontinuous. Instead, we use the method used by TORO to distribute over a chain of nodes the desired pose adjustment of the endmost node. In our case, the desired pose adjustment is given by temporarily applying \( \tilde{x}_c \) to \( z \) and normalizing the affected quaternions. The desired pose adjustment is the transform from \( b \)'s old pose to its new pose. This pose adjustment is distributed over the nodes from \( a \) down to \( b \), not including \( a \). As in TORO, we use the diagonal elements of \( B \) as the distribution weights. For details on this distribution algorithm, we refer the reader to [10]. This does not cause dog-legs, because \( \tilde{x}_c \) updates rotations even for position-only constraints.

F. Batch solving

It is possible to build a hybrid hessian \( H_c \) for updating a set \( C \) of constraints at once, by including the off-diagonal blocks of multiple constraints’ \( J_i J_i^T \).

\[
H_c = \sum_{c \in \mathcal{C}} J_i^T J_c + B_c
\] (35)

where the block-diagonal regularizer \( B_c \) is:

\[
B_c = \mathfrak{B}(H) - \sum_{i \in \mathcal{C}} \mathfrak{B}(J_i^T J_i)
\] (36)

The corresponding orthogonal decomposition form is:

\[
\begin{bmatrix}
\hat{J}_c \\
\Gamma_c
\end{bmatrix} \mathbf{x}_c =
\begin{bmatrix}
\mathbf{r}_c \\
\mathbf{0}
\end{bmatrix}
\] (37)

where \( \Gamma_c \) is created as before by the Cholesky decomposition \( B_c = \Gamma_c^T \Gamma_c \). We vertically stack \( J_c \) and \( \mathbf{r}_c \) corresponding to the constraints \( c \) in \( C \) to get \( J_c \) and \( \mathbf{r}_c \). When \( C \) contains all the constraints in the posegraph, (37) reduces to the full exact update equation (14), since \( J_c \) and \( \mathbf{r}_c \) become \( J \) and \( \mathbf{r} \) (see (8)), and \( \Gamma_c \) vanishes, since there are no constraints \( i \) such that \( i \notin \mathcal{C} \) (see (36)).

G. Special Considerations for GPS

In some instances, it is desirable to combine omitting nodes and batch-optimizing multiple constraints. For example, we may wish to solve for a locally exact update, by solving for only the nodes close to the robot position, as in [12]. This can be done in our system by omitting faraway nodes, and batch-optimizing all constraints that operate on nearby nodes. Another application is in the processing of GPS constraints. GPS constraints are characterized by long path sizes and large position residuals, and do not specify rotation. Relaxing a single GPS constraint \( c \) causes its path to bend in order to move the constrained node \( n \) closer to the desired position. Because \( c \) specifies no orientation for the node, \( n \) is free to rotate to align itself with the new direction of the path. This is harmful to convergence, as it rotates all of \( n \)'s subtree, increasing the residuals of other GPS constraints, which then do similar damage in turn. To avoid this, we update GPS constraints in batches. This eliminates spurious rotations by placing additional position constraints below \( n \) in the tree, preventing the sub-tree from bending away from them. We find that relatively small batch sizes are sufficient to prevent spurious rotations. For the Valencia and Paris datasets (section IV), we update GPS constraints in batches of 30 and 50, respectively. As when processing other constraints with long paths, we use interpolated solving to keep update times low.

H. Temperature

To aid convergence, we scale update \( x_c \) by a temperature parameter \( \tau \), before adding it to parameters \( z \) as: \( z \leftarrow z + \tau x_c \). We start with \( \tau = 1 \), and slowly decrease it over time. We do this by scaling \( \tau \) by 0.99 after each loop through all constraints. If a constraint \( c \)'s residual is large, the resulting \( \tau x_c \) may contain large rotation updates, which can adversely affect convergence. For such updates, we temporarily substitute \( \tau \) for a value \( \tau' \), which is chosen so that the largest rotation update in \( \tau' x_c \) does not exceed \( \pi/8 \).

I. Algorithm summary

We summarize our method in algorithm 1. The functions “Update” and “BatchUpdate” implement single and multi-constraint updates as described above, using interpolated solving to solve for no more than \( D_{\text{max}} \) nodes at a time.

IV. RESULTS

Fig. 3 shows a section of our “Paris1” posegraph before and after optimization, both with and without GPS constraints. We show a case where, without GPS constraints, the optimization causes rotations at an under-constrained intersection, causing the loop to rotate into an unrealistic

<table>
<thead>
<tr>
<th>Solver</th>
<th>Avg. time (s)</th>
<th>Max. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TORO</td>
<td>5598</td>
<td>2099</td>
</tr>
<tr>
<td>( D_{\text{max}} = 75 )</td>
<td>2.6435 * 10^{-4}</td>
<td>5.5595 * 10^{-2}</td>
</tr>
<tr>
<td>( D_{\text{max}} = 100 )</td>
<td>4.0791 * 10^{-4}</td>
<td>6.5095 * 10^{-2}</td>
</tr>
<tr>
<td>( D_{\text{max}} = 150 )</td>
<td>4.919 * 10^{-4}</td>
<td>9.5015 * 10^{-2}</td>
</tr>
<tr>
<td>( D_{\text{max}} = 200 )</td>
<td>5.812 * 10^{-4}</td>
<td>0.13747</td>
</tr>
<tr>
<td>( D_{\text{max}} = \infty )</td>
<td>2.0823 * 10^{-3}</td>
<td>3.583</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>nodes</th>
<th>edges</th>
<th>loop edges</th>
<th>max d</th>
<th>( D_{\text{max}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manhattan world</td>
<td>5300</td>
<td>5598</td>
<td>2099</td>
<td>184</td>
<td>80</td>
</tr>
<tr>
<td>Valencia w/o GPS</td>
<td>15031</td>
<td>15153</td>
<td>122</td>
<td>1161</td>
<td>40</td>
</tr>
<tr>
<td>Valencia w/GPS</td>
<td>15031</td>
<td>15440</td>
<td>409</td>
<td>1834</td>
<td>90</td>
</tr>
<tr>
<td>Paris1 w/o GPS</td>
<td>27093</td>
<td>27716</td>
<td>590</td>
<td>3599</td>
<td>190</td>
</tr>
<tr>
<td>Paris1 w/GPS</td>
<td>27093</td>
<td>28943</td>
<td>1817</td>
<td>3605</td>
<td>200</td>
</tr>
<tr>
<td>Paris2 w/o GPS</td>
<td>41957</td>
<td>55392</td>
<td>13384</td>
<td>701</td>
<td>150</td>
</tr>
<tr>
<td>Paris2 w/GPS</td>
<td>41957</td>
<td>56109</td>
<td>13878</td>
<td>802</td>
<td>200</td>
</tr>
</tbody>
</table>
configuration. We show that GPS constraints serve to limit such error. We also show an instance of the dog-leg problem experienced by TORO. Even though the dog-leg problem is typical with GPS constraints, it can also happen, as it did here, with loop-closing constraints that have a large position residual and small rotation residual. Our method does not suffer from this problem.

In fig. 5, we show the log-energy per constraint over time for our method and TORO. Our method reduces the error quicker, and converges to an average energy per constraint that is an order of magnitude lower than that of TORO. For our method, we use interpolated solving as described in section III-E, with different maximum values $D_{max}$ for the size of set $S_c$. GPS constraints were not used, to minimize dog-legs in TORO. The pose graph data was taken from a section of Valencia, Spain, with 15031 poses and 15153 constraints. The energy was measured after each loop through all constraints. In actual operation, only a few constraints are added or updated per frame, so the spacing of the points in the plot should not be interpreted as the required time per iteration. Rather, see table I for the average and maximum time per constraint for the same solvers and posegraph. The average constraint domain size was 1.53 poses, while the largest constraint domain was 1161 poses. The times shown are all within real-time bounds per frame, except when domain subsampling is turned off (by setting $D_{max} = \infty$).

Linearizing the relation between position error and rotation updates is necessary for properly addressing the dog-leg problem. However, such projective rotations can also cause oscillations in the face of excessive subsampling. To test our method’s robustness to oscillations, we ran the solver with various levels of subsampling, defined by $D_{max}$, the maximum number of nodes to solve for in (34). Table II shows the minimum values of $D_{max}$ which did not cause divergence. Note that these are not hard minimums, as divergence may also be avoided by lowering the initial temperature $\tau$ from 1.0. This table is only intended to illustrate the potential danger of over-subsampling. Table II also shows each map’s number of nodes, edges, loop edges, and “max $d$”. Loop edges are edges with more than two nodes in their path (they are also counted under “edges”). The “max $d$” is the length of the longest edge path in the map. The “Manhattan world” dataset was originally used by Olson in [9] (see fig. 6a).

In fig. 6, we show some maps before and after solving with our method. The “before” images show the poses as initialized by starting at the root of the parametrization tree, and crawling downwards, concatenating the tree edges’ transforms. For a pose graph with no loop closures, this would be equivalent to dead-reckoning. The red edges are edge residuals, connecting the desired pose of a node to its actual pose. Relative constraints’ residuals connect two poses, while GPS constraints’ residuals connect a pose to a spot in empty space, indicating the desired position. Redder residuals indicate higher error. Longer residuals do not necessarily have higher error, as some edges are less stiff than others. In particular, GPS constraints are much less stiff than other types, due to GPS’ imprecision. Our method performs well on graphs with ample loop closures, such as Olson’s “Manhattan world”, converging to an average energy per constraint of 1.596, compared to TORO’s 2.062. Our method completely collapses most relative constraint residuals (fig. 6b, 6d), and greatly reduces GPS residuals (fig. 6f). A small number of lines which overlap in fig. 6e can be seen to have split apart in 6f. These are parallel runs where the loop closure detector failed to recognize as traversing the same path, and therefore did not connect together with a constraint. Outdoor urban maps can have fewer loop-closures, due to the difficulty of detecting them in highly dynamic environments. Despite this distortion, the solved map aligns relatively well.

Fig. 6. Solved maps Pose graphs, before and after 10 iterations with $D_{max} = 200$. Pose graph sizes are given in table II. Initial configurations show the poses as set by concatenating constraint transforms down the tree, as described in section III-A. Constraint residuals are shown as brown/red lines connecting the constraint’s desired pose to the actual pose. Brighter red indicates higher error. Valencia (fig. 6c, fig. 6d) is shown at an oblique angle, to better show its error residuals, which are primarily vertical.

Fig. 7. Montmartre, Paris An overlay of the converged poses of fig. 6f on a satellite image from Google Earth.
On each iteration, our method efficiently updates a subset of the pose graph, in a manner that approximates the effects of nodes and constraints outside of that set. There is considerable flexibility in how to choose this set, affording several avenues of future investigation. One is constraint prioritization, where an effort is made to update constraints with large error more frequently than those with small error, for quicker convergence. Another is node prioritization, where instead of subsampling a constraint domain uniformly, we select nodes according to the the needs of the application. An example would be to prioritize local nodes during real-time exploratory SLAM. This is an approach similar to [12], but with the added benefit of maintaining global consistency. Another example is to add loop-closing to visual odometry by treating bundle adjustment as a local batch update, within a SLAM framework.

VI. ACKNOWLEDGMENTS

The authors thank Google for the pose graph data, Mike Kaess for Olson’s “Manhattan world” dataset, and the reviewers for the helpful comments.

REFERENCES