MSGD: Scalable Back-end for Indoor Magnetic Field-based GraphSLAM

Chao Gao* and Robert Harle*

***Abstract*— Simultaneous Localisation and Mapping (SLAM) systems that recover the trajectory of a robot or mobile device are characterised by a front-end and back-end. The front-end uses sensor observations to identify loop closures; the back-end optimises the estimated trajectory to be consistent with these closures. The GraphSLAM framework formulates the back-end problem as a graph-based optimisation on a pose graph.**

This paper describes a back-end system optimised for very dense sequence-based loop closures. This arises when the front-end generates magnetic loop closures, among other things. Magnetic measurements are fast varying, which is good for localisation, but the requirement for high sampling rates (50 Hz+) produces many more loop closures than conventional systems. To date, however, there is no study optimising GraphSLAM back-end for sequence-based magnetic loop closures. Hence we introduce a novel variant of the Stochastic Gradient Descent-based SLAM algorithm called MSGD (Magnetic-SGD). We use high-accuracy groundtruth system and extensive real datasets to evaluate MSGD against state-of-the-art back-end algorithms. We demonstrate MSGD is at least as good as the best competitor algorithm in terms of quality, while being faster and more scalable.

I. Introduction

Joint estimation of the trajectory of a robot (or mobile device) and the map of an unknown environment is the core idea of Simultaneous Localisation and Mapping (SLAM) [21]. SLAM systems find the most probable trajectory and/or the model of the environment given the sensor observations. This problem is often formulated as an optimisation problem on a pose graph—GraphSLAM is the dominant framework, where features and/or robot poses are represented by nodes and constraints resulting from observations or odometry/pedometry are modelled by edges. A SLAM system is typically split into two components: the *front-end* (which identifies constraints) and the *back-end* (which optimises the trajectory/map given those constraints).

Conventional SLAM systems are based on vision, which brings with it high computational costs. They typically produce a low density of constraints that the back-end must then optimise for. In this paper we present a GraphSLAM back-end that can handle a far greater density of constraints.

Our motivation for this is the use of magnetic signals in the front-end. Magnetic signals are attractive because they are ubiquitous and typically have rapid spatial variation indoors, which is ideal for accurate localisation. The sensors are also small, inexpensive and readily available. However, this fast variation demands fast sampling (50 Hz+), which produces dense sequence-based loop closures for the back-end. Although we focus on magnetic data here, we emphasise that the back-end we introduce is applicable for any signal producing a high density of sequence-based constraints. We make following contributions in this paper:

- We describe the design of an appropriate SLAM back-end, which uses a modified stochastic gradient-descent method (SGD [20]) we call MSGD;
- We use extensive real datasets to evaluate MSGD against state-of-the-art GraphSLAM algorithms on optimising the pose graph with sequence-based magnetic loop closures;
- We open source our implementation of MSGD with all the datasets (`https://github.com/chaogao-cam/MSGD`).

II. Related Work

SLAM Applications. SLAM techniques were developed for robot navigation [21], where loop closures are conventionally derived from feature tracking in vision [18]. They have also been applied to tracking people using pedestrian dead reckoning (PDR) [5]. In this context, video is unattractive due to privacy concerns.

Other signals have been investigated, most notably WiFi. *WiFi GraphSLAM* [11] uses WiFi signal strength data to optimise pedestrian trajectory. It achieved a mean accuracy of 2.23 m[1] using pedometry and a GraphSLAM formulation.

Magnetic SLAM. Magnetic signals can be used to generate loop closures by analysing temporal *sequences* of measurements (a single magnetic measurement is not spatially unique). This has been demonstrated by Jung et al. in robotics [12] and by the authors in PDR [8]. In both cases sub-metre accuracy was achieved, although back-end efficiency was not considered.

Our technique [8] is used to generate loop closures to test our back-end. We take windows of the magnetic field vector and search for similar magnetic windows in the history of the path (e.g. Figure 1). The detection of a **loop closure** introduces a group of constraints (**loop closure constraints** herein). The back-end then applies the GraphSLAM framework to estimate the trajectory most consistent with all the constraints. Because of the high frequency of magnetic measurements, a 10-minute trajectory can generate a graph with 30,000+ nodes, making back-end efficiency very important.

*Chao Gao and Robert Harle are with the Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK `cg500@cam.ac.uk and rkh23@cam.ac.uk`

[1]The metric used to compute the accuracy is the *subjective-objective* error, which measures how accurately the resultant trajectory meets the constraints. Please refer to [11] for more details.

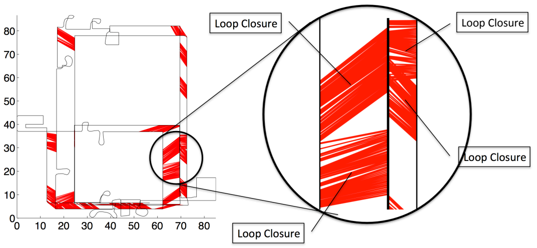


Fig. 1: Each loop closure gives rise to a set of loop closure constraints. The magnified portion shows the loop constraints (red lines) derived from four distinct loop closures.

GraphSLAM. The GraphSLAM framework formulates the SLAM problem as a graph optimisation problem. Let C be the set of all constraints in the pose graph, X be the state vector of all the poses. The chi-squared (χ^2) errors of the whole pose graph can be formulated as

$$F(X) = \sum_{i \in C} F_i = \sum_{i \in C} e_i^T \Omega_i e_i \tag{1}$$

where Ω_i is the information matrix that encodes the constraints between poses of a constraint i and and e_i is the residual error of this constraint. The GraphSLAM goal is to find a configuration of X (X^*) that minimises F i.e. $X^* = \arg\min_X F(X)$.

Many algorithms utilise the sparsity of the information matrix of the pose graph to efficiently optimise the graph [22], [4]. However, they irrevocably introduce linearisation error that can lead to poor estimates. Lu and Milios [16] suggested a brute-force nonlinear least squares implementation to optimise the pose graph, which iteratively solves a linear system of size proportional to the number of nodes. More efficient approaches have since been introduced: Gauss-Seidel relaxation [3], [6]; Stochastic Gradient Descent (SGD) [20]; tree-parameterised SGD [9]; and the use of modern linear solvers [17], [13], [14], [15].

III. State-of-the-art GraphSLAM Solvers

We compare our algorithm to three state-of-the-art GraphSLAM solvers: g2o [15], SGD [20], and Toro [9]. All three linearise the error e_i at the current state X to get

$$F_i(X + \Delta X) = e_i^T \Omega_i e_i + 2 e_i^T \Omega_i J_i \Delta X + \Delta X^T J_i^T \Omega_i J_i \Delta X \tag{2}$$

where J_i is the Jacobian of e_i at the current state. At the minimum $X^* = X + \Delta X$.

g2o This is a conventional non-linear least square method that iteratively linearises the error F at current state X. It computes $\Delta X = -(J^T \Omega J)^{-1} J^T \Omega e$ (which is derived by differentiating Equation 1 with respect to ΔX and setting the result to zero). It uses modern linear solvers such as sparse Cholesky decomposition to solve the linear system efficiently. The prerequisite is that $J^T \Omega J$ is sparse and positive definite.

SGD An iterative solver that computes ΔX based on a randomly-selected *single* constraint. The update equation becomes:

$$X^* = X + \lambda H^{-1} J_i^T \Omega_i e_i \tag{3}$$

where $H = J^T \Omega J$, and λ is a learning rate that decreases with each iteration. λ allows the system to converge to an equilibrium point when antagonistic constraints exist. H acts as a pre-conditioner to scale and distribute errors according to the importance of each constraint and node. For simplicity in the inversion it is approximated as $H \simeq diag(H)$. Solving a given constraint distributes weighted residual errors to a sequence of consecutive poses. For example, if a constraint connects two poses with indices a and b respectively, then solving this constraint distributes the error among poses $[a+1, a+2, ...b]$. Thus a naïve implementation requires $O(N)$ (assume the graph contains N poses) time to perform the update of a single constraint. A special binary tree was proposed to speed up this process to $O(logN)$ time [20], [19]. We describe this in more detail later since we extend it for our system.

Toro The Toro algorithm is based on the same principle as SGD, but it adopts a tree parameterisation of the pose graph. With this parameterization, the number of nodes involved in the update of each constraint depends only on the topology of the environment, and the interactions between antagonistic constraints are kept small. Toro will typically converge much faster than SGD.

IV. The MSGD Algorithm

Our tests with the three algorithms outlined above revealed that they did not scale well with increasing constraints. We therefore developed a novel variant (MSGD) that adapts SGD to handle large numbers of (sequence-based) constraints efficiently. A viable alternative would be to focus on reducing the number of nodes and edges (constraints), either by discarding samples or using graph sparsification techniques ([2]). However, the dense magnetic loop closure constraints provide very rich information about how parts of the pose graph should be stretched, compressed or aligned, and any sparsification is only likely to reduce the trajectory accuracy. We believe it is better to optimise the back-end first.

A. Optimising the Pre-conditioner (H) Calculation

1) Optimising by Tree Operation: The SGD method requires us to compute H in equation 3 at each iteration (lines 6 to 15 of Algorithm 1). We assume a graph with N nodes and E constraints (edges). Then the cost to compute H is $O(EN)$. The original SGD computes M only at iterations 1, 2, 4, 8, ... to reduce the cost [20]. But for our dense graph E can be seen as $O(N)$ so the cost is nearly $O(N^2)$, where N can be 30,000+ for only a 10-minute trajectory (Section I). Therefore, the time complexity needs to be further reduced for scalability. We have optimised the original SGD algorithm (Algorithm 1) for our dense graph as shown in Algorithm 2. We briefly analyse both algorithms for comparison.

Algorithm 1 Pre-Conditioner Estimation in SGD

```
1: $iters = 0$
2: loop
3:     $iters$++
4:     ...
5:     // Compute $M = diag(H) = diag(J^T \Omega J)$
6:     $M = zeros(numPoses, 3)$
7:     for all $a, b, t_{ab}, \Omega_i$ in $Constraints$ do
8:         $pose_a$ = getPose(a)                    // $O(logN)$
9:         $R$ = rotation matrix of $pose_a$
10:        $W = R\Omega_i R^T$
11:        for $i$ = a+1 to $b$ do
12:            $M_{i,1:3} = M_{i,1:3} + diag(W)$
13:            ...
14:        end for
15:    end for
16:
17:    // Modified Stochastic Gradient Descent
18:    for all $a, b, t_{ab}, \Omega_i$ in $Constraints$ do
19:        ...
20:        $totalWeight = zeros(1, 3)$
21:        for $i$ = a+1 to $b$ do
22:            $totalWeight = totalWeight + 1/M_i$
23:        end for
24:        ...
25:    end for
26: end loop
```

Algorithm 2 Improved Pre-Conditioner Estimation

```
1: $iters = 0$
2: loop
3:     $iters$++
4:     ...
5:     // Compute $M = diag(H) = diag(J^T \Omega J)$
6:     $M = zeros(numPoses, 3)$
7:     {Compute all poses in linear time}
8:     for all $a, b, t_{ab}, \Omega_i$ in $Constraints$ do
9:         $pose_a$ = getPose(a)          // This is $O(1)$ now
10:        $R$ = rotation matrix of $pose_a$
11:        $W = R\Omega_i R^T$
12:        // Distribute values in $O(log(N))$ time by tree
13:        {Distribute $diag(W)$ over $M_i, i \in [a+1, b]$}
14:        ...
15:    end for
16:
17:    {Recover $M_i, i \in [1, numPoses]$ in O(N) by tree}
18:
19:    // Compute cumulative weight
20:    $cmWeight_1 = 1/M_1$
21:    for $i$ = 2 to $numPoses$ do
22:        $cmWeight_i = cmWeight_{i-1} + 1/M_i$
23:    end for
24:
25:    // Modified Stochastic Gradient Descent
26:    for all $a, b, t_{ab}, \Omega_i$ in $Constraints$ do
27:        ...
28:        $totalWeight = cmWeight_b - cmWeight_a$
29:        ...
30:    end for
31: end loop
```

The original algorithm (Algorithm 1) computes M (which is used to approximate H) with E calls to function $getPose$. Given two poses, a and b, and a constraint between them, it will distribute weighted errors across the poses $[a+1, a+2, ...b]$. This can be done naïvely in $O(N)$ but Olson showed how to use a binary tree to speed up this process [19]. Each node i ($0 \leq i < N$) of the tree holds a *node value* n_i and maintains a pose value v_i. The pose value v_i is defined as the sum of the node value n_j from node i up to the root of the tree along the ancestry chain. Each v_i can be computed in $O(logN)$ time, and adding some amount to each v_i with $i \geq I$ (where I is an arbitrary index and $0 \leq I < N$) can be done in $O(logN)$ time. Thus $getPose$ can be computed in $O(logN)$. The cost to compute M is then given by E calls to $getPose$ (costing $O(ElgN)$ and E calls to a $O(N)$ subloop (lines 11 to 14 of Algorithm 1). The total cost is then $O(EN + ElgN) = O(EN)$.

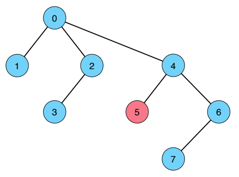


Fig. 2: Example of linear-time pose computation.

Our optimised algorithm (Algorithm 2) speeds up both the $getPose$ and the $O(N)$ subloop. First we compute all of the poses *before* the estimation of H. Recall that each node of the tree represents a pose in the graph (and holds the pose value v_i), and computing a pose value is to sum the node value n_j from node i up to the root along the ancestry chain. Taking the tree shown in Figure 2 as an example, the pose value v_5 is the sum of node values n_5, n_4 and n_0, i.e. $v_5 = n_5 + n_4 + n_0$. Similarly, we have $v_4 = n_4 + n_0$. Then we can get $v_5 = n_5 + v_4$—i.e. computing a pose value v_i can be seen as adding the node value n_i to its parent node's pose value. Therefore we compute the poses using a breadth first search, which ensures that the parent pose is always computed before the child pose. This requires a $O(N)$ step (line 7 of Algorithm 2) but thereafter makes $getPose$ $O(1)$.

By careful inspection, we find that when estimating M, each constraint causes the same amount of error distributed to a continuous part of the diagonal elements of H (Line 11 to 14 of Algorithm 1). This is a similar process to updating the poses when solving a constraint and we therefore apply the tree-based mechanism to maintain the elements of M and speed up the M estimation. Now the subloop is replaced with a $O(lgN)$ process (line 13 of Algorithm 2). The total cost to compute M (H) is thus $O(N) + O(E(lgN + lgN)) = O(ElgN)$ vs the original $O(EN)$.

Similar savings apply to the gradient descent. In the original algorithm M is used to compute $totalWeight$ for each constraint (Line 17 to 23). Each $totalWeight$ requires $O(N)$ time to compute, so a single iteration requires $O(EN)$ time to compute $totalWeight$ for all the constraints.

In the optimised algorithm, M is maintained using a tree, which allows us to speed the process up. We first recover each element of M ($M_i, i \in [1 : numPoses]$) in linear time. Then, define $cmWeight_i = \sum_{j\in[1,i]} 1/M_j$, so that the $totalWeight$ for each constraint can be computed in $O(1)$ time as shown in Algorithm 2. Now, each iteration needs only $O(N+E)$ time to compute $totalWeight$ for all constraints, versus the original $O(EN)$

2) Optimising by Approximation: We can speed up the estimation of H by exploiting the fact that each loop closure results in a group of loop constraints. In Algorithms 1 and 2 we compute $W = R\Omega_i R^T$ for each constraint to estimate M. W is the information matrix of the constraint in the global reference frame, which takes two matrix operations to compute. However, we can use the W of any constraint in a group to approximate the W for the other constraints in that same group. Since a typical group contains hundreds of constraints in our context, this is a significant saving.

B. *Error Propagation Over Selected Constraints Only*

MSGD reduces the processing cost by considering a mini-batch of all constraints at once. More specifically, it considers a single loop closure at a time, solving only a small selected subset of the hundreds of loop constraints it introduces. In fact, only two or three constraints are selected per group and the selection is based on the type of loop closure. Figure 3 shows how loop closures are classified as either Type I or Type II based on the order (ascending or descending) of the pose indices.

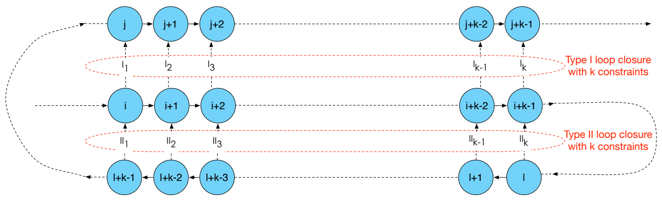


Fig. 3: Loop closure classification. The pose graph showed here is typical for the moving trajectory in a corridor. Two loop closures are shown: a Type I loop closure with k constraints I_1 to I_k; a Type II loop closure with k constraints II_1 to II_k. The rule to classify loop closures is: write the index number of poses in the older sequence (i.e. the pose sequence with smaller index numbers) in ascending order (i.e. $i, i+1, ...i+k-1$), if the index number of poses in the later sequence (i.e. the pose sequence with larger index numbers) is increasing (i.e. $j, j+1, ..., j+k-1$), then this is a Type I loop closure; otherwise (i.e. $l+k-1, l+k-2, ..., l$), this is a Type II loop closure.

Solving a constraint in SGD is equivalent to distributing the error over a consecutive sequence of poses. For example, in Figure 3, solving the constraint I_1 in Type I loop closure spreads the error of I_1 over $Pose_{i+1}$, $Pose_{i+2}$,..., ,$Pose_{j-1}$, $Pose_j$. Similarly, solving the constraint II_1 in Type II loop closure spreads the error of II_1 over $Pose_{i+1}$, $Pose_{i+2}$,..., $Pose_{l+k-2}$, $Pose_{l+k-1}$. In Figure 4, we draw the poses affected by solving a certain constraint on each row, and align the poses with the same indices vertically, to emphasise that solving constraints within the same loop closure group affects many common poses. For instance, solving I_1 in the Type I loop closure affects $Pose_{i+1}$ to $Pose_j$ and solving I_2 affects $Pose_{i+2}$ to $Pose_{j+1}$. So, the poses affected by solving I_1 and solving I_2 respectively have $Pose_{i+2}$ to $Pose_j$ overlapped.

The loop closure constraints within a loop closure group can be consistent or antagonistic to each other, and the poses affected by different constraints in the same loop closure group overlap heavily. So solving a constraint can affect the states of other constraints within the same loop closure group. For example, solving a constraint C may cause other constraints consistent with C to be solved or nearly solved, but it might cause antagonistic constraints to be pushed off their solved states (if they have been solved or nearly solved). SGD uses a decreasing learning rate to modulate the interactions between constraints. During an full iteration, each constraint within a loop closure group will be solved. In contrast MSGD identifies a small subset of constraints within each loop closure group that can be used to give a good result without requiring that *all* group constraints be solved.

Our constraint selection process is as follows. For a group of constraints, first solve the constraint with the max χ^2 error (denote this constraint as C_{max}). This step requires us to compute the χ^2 errors for all constraints and find the one with the max χ^2 error. The χ^2 error e of a constraint is:

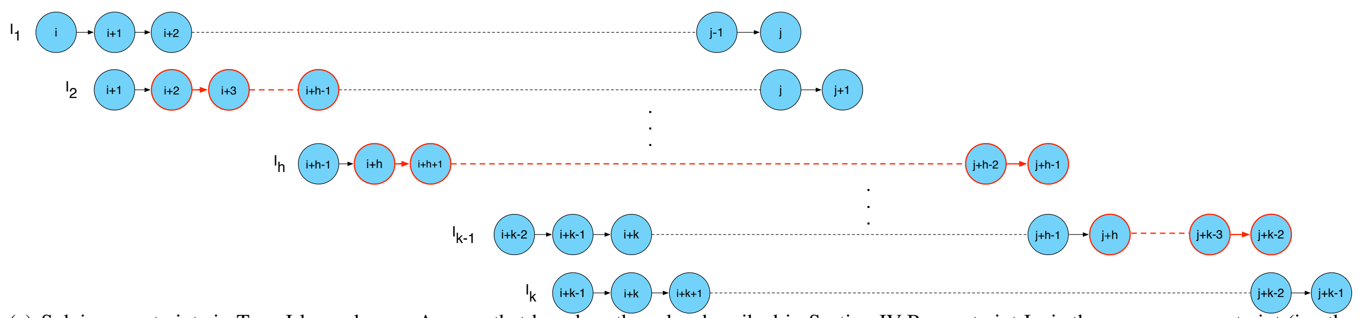
$$e = r^T W r \simeq r^T r \tag{4}$$

where r is the residual error of this constraint and $W = R\Omega_i R^T$, which is assumed constant for a given group as per Section IV-A.2. Please note that approximating e as $r^T r$ is not appropriate in theory because different components of r need to be scaled properly. But this makes little difference to the converged results we tested.

Next, we solve a very small set of constraints that are consistent with C_{max}. We define the consistency between two constraints as: a constraint C_a with residual error r_a is consistent with another constraint C_b with residual error r_b when $r_a * r_b > 0$[2]. We then select the set of constraints that needs to be solved based on these rules:

1) All constraints in this set are consistent with C_{max};
2) The constraints in this set cover as many poses as possible;

[2]Here we assume r is a scalar for simplicity. In practice, r is a vector (e.g. $r = x, y, heading$) and solving a constraint requires distributing errors separately for each component. In this case, our method can be applied in the same way for each component.


(a) Solving constraints in Type I loop closure. Assume that based on the rules described in Section IV-B, constraint I_h is the max error constraint (i.e. the C_{max}), constraint I_2 and I_{k-1} are the selected constraints (i.e. C_{I_a} and C_{I_b} respectively). When solving I_2 and I_{k-1}, the bounded residual errors are distributed among only the poses that not covered by I_h (i.e. $Pose_{i+2}$ to $Pose_{i+h-1}$ and $Pose_{j+h}$ to $Pose_{j+k-2}$).

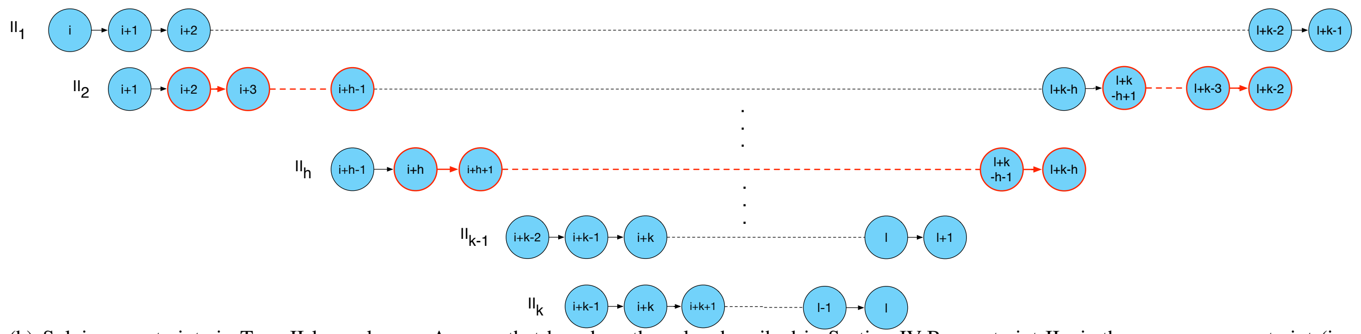

(b) Solving constraints in Type II loop closure. Assume that based on the rules described in Section IV-B, constraint II_h is the max error constraint (i.e. the C_{max}), constraint II_2 is the selected constraint (i.e. the C_{II}). When solving II_2, the bounded residual errors are distributed among only the poses that not covered by II_h (i.e. $Pose_{i+2}$ to $Pose_{i+h-1}$ and $Pose_{l+k-h+1}$ to $Pose_{l+k-2}$).

Fig. 4: Illustration of solving selected constraints in Type I and Type II loop closure respectively.

3) This set contains as few constraints as possible.

Solving these selected constraints can push the solution in the same direction, and the conflicting effects of antagonistic constraints can be attenuated. Since very few constraints need to be solved for a loop closure group, the computational cost can be lowered. The selection process is trivial:

- For the Type I loop closure, loop from the first constraint to C_{max} until a constraint that is consistent with C_{max} is found (denoted by C_{I_a}); then loop from the last constraint to C_{max} until a consistent constraint is found (denoted by C_{I_b}).
- For the Type II loop closure, loop from the first constraint to C_{max} until a consistent constraint is found (denoted by C_{II}).

Thus, when solving the Type I loop closure, we first solve C_{max}, and then solve C_{I_a} and C_{I_b}; when solving the Type II loop closure, we first solve C_{max} and then C_{II}. However, it should be noted that, when solving C_{I_a}, C_{I_b} and C_{II} (after C_{max} is solved), we do not distribute errors over the poses that overlapped with the poses affected by C_{max}, because this will disturb the solved state of C_{max}. Instead, we distribute errors only over the poses that are not covered by C_{max} (Figure 4). To avoid over optimisation we bound the residual errors conservatively before distribution. Algorithm 3 shows how the residual error of C_{I_a} is bounded and distributed over specific poses after C_{max} is solved when solving the Type I loop closure. More specifically, we distribute the residual error e_r over poses with indices $a+1, a+2, ..., a_{max}$ (the poses covered by C_{I_a} but not by C_{max}). Here we briefly described how e_r is determined. Because the amount of error distributed to a pose is proportional to the weight of this pose, we can determine how much error of a constraint is distributed over a specific pose sequence. We define e_{r1} as the error e of C_{I_a} minus the amount of error distributed by C_{max} over the poses covered by both C_{max} and C_{I_a} (Line 19). Define e_{r2} as the portion of e that should be distributed over poses only covered by C_{I_a} but not by C_{max} (Line 24). Then e_r is set to one of e_{r1} and e_{r2} with smaller absolute value (Line 25)

The method to solve C_{I_b} and C_{II} is similar so omitted here. In summary, instead of solving all the constraints in a loop closure group, only two or three selected ones are solved so as to greatly reduce the computational cost.

V. Evaluation

To evaluate the performance of MSGD we compare it to open source implementations of g2o[3], Toro[4], and SGD. We implemented the latter using C++ to provide a direct comparison with MSGD. The SGD implemented here randomly selected a constraint to process each time (as in [19]) rather than iterating through all constraints in a fixed order (as in [20]) because we found randomisation (which is better for escaping local minima) gave much better performance for all our datasets.

[3]https://openslam.org/g2o.html
[4]https://www.openslam.org/toro.html

Algorithm 3 Error Propagation

```
1:  $iters = 0$
2:  **loop**
3:      $iters$++
4:      ...
5:      // *Modified Stochastic Gradient Descent*
6:      // *Each loop closure introduces a mini-batch (group) of constraints into the system*
7:      **for all** Mini-batch of Constraints **do**
8:          {Find max error constraint $C_{max} = \{a_{max}, b_{max}, e_{max}, t_{max}, \Omega_{max}\}$}
9:          $w_{max} = cmWeight_{b_{max}} - cmWeight_{a_{max}}$
10:         Solve $C_{max}$
11:         // *Find and solve* $C_{I_a}$
12:         **for** $C = \{a, b, e, t_{ab}, \Omega_i\}$ from first constraint to $C_{max}$ **do**
13:             **if** !consistent($C$,$C_{max}$) **then** continue; **end if**
14:             $w_{ovlp} = 0$            // *the overlapped weight*
15:             **if** $b > a_{max}$ **then**
16:                 $w_{ovlp} = cmWeight_b - cmWeight_{a_{max}}$
17:             **end if**
18:             $e_{ovlp} = e_{max} * w_{ovlp} / w_{max}$
19:             $e_{r1} = e - e_{ovlp}$
20:             // *Check consistency again*
21:             **if** $e * e_{r1} < 0$ **then** continue; **end if**
22:             $totalWeight = cmWeight_b - cmWeight_a$
23:             $w_{residual} = totalWeight - w_{ovlp}$
24:             $e_{r2} = e * w_{residual} / totalWeight$
25:             $e_r = (|e_{r1}| < |e_{r2}|) ? e_{r1} : e_{r2}$ // *Bound error*
26:             $W = R\Omega_i R^T$
27:             $d = 2We_r$
28:             $\lambda \propto \frac{1}{iters}$            // *the learning rate*
29:             $\beta = (a_{max} - a) * \lambda * d$
30:             **if** $|\beta| > |e_r|$ **then** $\beta = e_r$ **end if**
31:             {Distribute $\beta$ over $pose_{a+1}$ to $pose_{a_{max}}$}
32:             break;
33:         **end for**
34:
35:         // *Find and solve* $C_{I_b}$ ...
36:     **end for**
37: **end loop**
```

A. Datasets

We collected a series of datasets using pedestrians carrying smartphones in different testbeds, which are described in detail in [7], [8]. Table I gives statistics on the various datasets (where each 'pose' is associated with a magnetic measurement). Note that high accuracy (3cm) location ground truth ([1]) was available in the WGB2a testbed. Figure 5 illustrates two sample inputs to the back-end: each detected and validated loop closure constraint (thus we assume no false positive ones) from the front-end is indicated by a red line.

Please note that we do not use relative motion constraints for SGD and MSGD since this overconstrains the system.

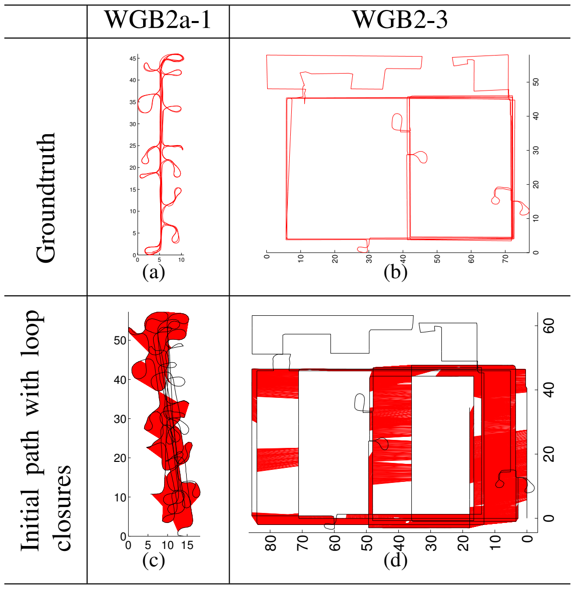


Fig. 5: Two sample datasets (WGB2a-1 and WGB2-3) used for evaluation of SLAM back-end algorithms.

However, both g2o and Toro require these constraints. For SGD (and MSGD), it adopts an incremental state space so it does not rely on the relative motion constraints to maintain the topology of the trajectory. For g2o, it needs the relative motion constraints to keep the information matrix positive-definite and to maintain the trajectory topology. For Toro, the special tree parameterisation it adopts requires the relative motion constraints to keep the connectivity of the input pose graph thus to ensure the success of its tree building process.

TABLE I: Statistics of all datasets

Dataset	Poses	Relative Motion Constraints	Loop Closure Constraints	Loop Closures
WGB2a-1	26262	26261	10382	29
WGB2a-2	3982	3981	596	2
WGB2a-3	8177	8176	996	4
WGB2a-4	20671	20670	7221	29
WGB1-1	18551	18550	1370	26
WGB1-2	25958	25957	16213	109
WGB1-3	14639	14638	2820	19
WGB1-4	13353	13352	1338	9
WGB1-5	29941	29940	790	5
WGB2-1	7490	7489	152	5
WGB2-2	15692	15691	2690	27
WGB2-3	17645	17644	9690	130
WGB2-4	16195	16194	401	5
ENG-1	4314	4313	237	4
ENG-2	12748	12747	1125	7
RUTH-1	7218	7217	7200	60
RUTH-2	7980	7979	4117	23
KX-1	16402	16401	1165	20
KX-2	21499	21498	6094	41

B. Metrics

To evaluate the performance of back-end algorithms quantitatively, we adopt the **State-Squared Error** (SS_{error}) proposed in [19] as the metric. For a graph with N poses, denote the configuration of the result trajectory by

X, where $X = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$, and denote the groundtruth configuration by $\bar{X}$, where $\bar{X} = \{(\bar{x}_1, \bar{y}_1), (\bar{x}_2, \bar{y}_2), ..., (\bar{x}_N, \bar{y}_N)\}$. Then :

$$SS_{error} = mean\{dis_i^2 \mid dis_i^2 = (x_i - \bar{x}_i)^2 + (y_i - \bar{y}_i)^2\} \quad (5)$$

To give an unambiguous comparison, rigid-body transform T that minimises SS_{error} is computed and applied to either the result trajectory or the groundtruth trajectory beforehand [19]: we compute T using the algorithm in [10].

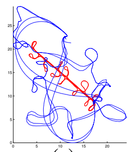
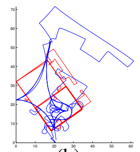
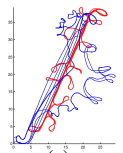
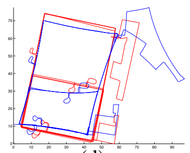
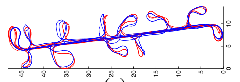
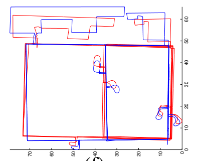
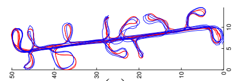
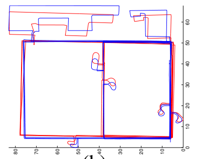
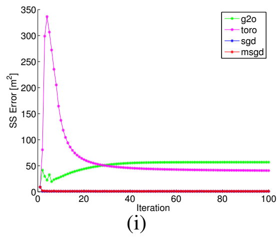
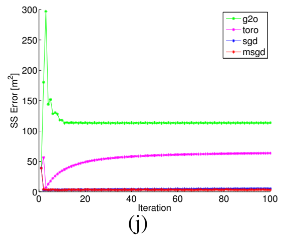
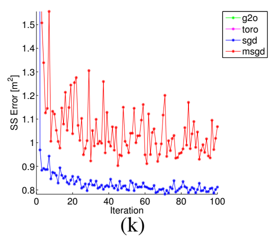
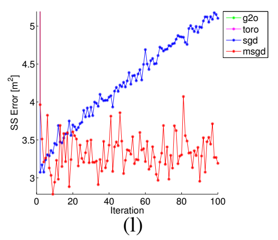
	WGB2a-1	WGB2-3
g2o	 (a)	 (b)
Toro	 (c)	 (d)
SGD	 (e)	 (f)
MSGD	 (g)	 (h)
SS Error	 (i)	 (j)
Zoomed-In SS Error	 (k)	 (l)

Fig. 6: Sample SLAM results. The trajectories (the red is the transformed groundtruth and the blue the slam result) and the SS errors are the results of the 100th iteration.

C. Results and Analysis

We ran 100 iterations of g2o, Toro, SGD and MSGD on each dataset. Sample results are shown in Figure 6. [5] We found that g2o and Toro failed to converge to acceptable results in most cases, but SGD and MSGD worked well on each dataset. As described before, all the algorithms explicitly minimise the system χ^2 error defined in Equation 1, and the χ^2 error of all the algorithms would converge to zero after enough iterations. However, whether an algorithm converges to the minimum SS error (a result that has the best quality from a practical perspective) depends on how it approached the minimum χ^2 error. Generally speaking, the methods based on Cholesky decomposition like g2o can find the lowest χ^2 error because they usually take the path leading directly to it. However, these methods can easily get stuck in local minima because the state space of the non-linear optimisation problem have long valleys with small χ^2 error. Randomised algorithms like SGD and MSGD are capable of escaping local minima and finding the path leading to the global minimum. Despite Toro having a very similar error distribution mechanism to SGD, it iterates over each constraint in a fixed order (due to its tree parameterisation), making it much less randomised in practice.

For most of the datasets tested, MSGD gave equivalent or better trajectories to SGD does (according to the final SS errors). The difference between the resultant trajectories of both algorithms is subtle. Note that the SS errors of both SGD and MSGD vary within a small range after convergence (Figure 6(k) and 6(l)). This is because both of them take stochastic steps in the state space around a minimum. However, the amplitude of the fluctuations in the SS errors of MSGD is larger than that of SGD because MSGD is 'more randomised' than SGD. SGD simply solves every constraint in the system, while MSGD solves only a few selected constraints based on their importance. In different iterations MSGD solves different constraints, which causes different changes to the SS error, and that is where the large fluctuation comes from. So long as the amplitude of the fluctuation is within a small range (not exceeds 1 m^2 in most cases), the final result is still stable and approaches the groundtruth.

In terms of scalability, however, MSGD has the advantage over SGD. Figure 7 illustrates how the iteration time changed with constraint number for the algorithms. For comparison, we include the results for a version of MSGD that uses only the optimisations described in Section IV-A (i.e. no constraint selection). This variant is labelled MSGD$^-$. We see that even without the constraint selection, MSGD is notably more scalable. With the constraint selection, full MSGD gains up to 40% improvements in execution time compared with MSGD$^-$. Please note that this improvement also increases as the number of constraints increases, which demonstrates the scalability of MSGD. The full MSGD algorithm is 3 to 8 times faster than SGD and the time per

[5]Please refer to Chapter 5 of [7] for detailed results and analysis of all the datasets.

iteration of MSGD increases much more slowly than that of SGD when the number of constraints is increasing. In fact, the computational cost of MSGD is proportional to the number of loop closures but not the number of constraints, giving it a significant advantage in the context of the high-frequency magnetic measurements.

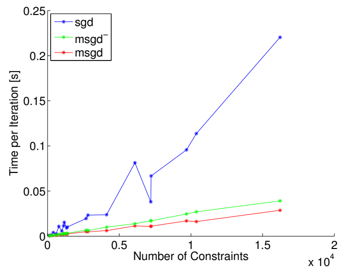


Fig. 7: CPU time per iteration of SGD, MSGD$^-$ and MSGD (by running each algorithm on every dataset in Table I). Relative motion constraints are discarded.

We also found that SGD exhibited slight divergence on WGB2-3 (Figure 6(l)) and nine other datasets (WGB2a-2, WGB2a-3, WGB2a-4, WGB1-2, WGB1-3, WGB2-2, WGB2-4, RUTH-1 and RUTH-2). MSGD did not have the same problem despite a larger amplitude of fluctuation. This can again be attributed to the constraints selection strategy of MSGD (Section IV-B). By solving only the most important constraints it keeps the interaction between antagonistic constraints small, achieving better consistency in the system states. So, the solution will not be pushed away from the minimum area. However, SGD simply solves every constraint within a loop closure group regardless their impacts to the stability of the system states, which causes divergence in these results.

VI. Conclusion and Future Work

We have evaluated the performance of three state-of-the-art SLAM back-end algorithms (g2o, Toro, SGD) on solving the pose graphs with magnetic sequence-based constraints. We demonstrated SGD achieves the best performance but is not optimised for the specific problem. We then introduced a novel variant of the SGD algorithm that is optimised for magnetic sequence SLAM, MSGD. It has been demonstrated that although MSGD has slightly larger fluctuations in the SS errors (after convergence) than SGD, they achieve equivalent trajectory quality. MSGD is also found to be more efficient and scalable when dealing with much larger datasets. In addition, MSGD does not have the problem of divergence as the SGD does. Therefore, for large magnetic sequence SLAM, we believe MSGD is a better solution.

We should note that the prerequisite for magnetic field-based SLAM systems is that the trajectory contains long loop closures in order for robust sequence-based matching. But compared with more flexible systems like the vision-based SLAM, the cost of magnetic field-based SLAM is much lower. So, it could be a complementary technique integrated into other SLAM systems. We leave this for future work.

References

[1] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a sentient computing system. *IEEE Computer*, 34(8), August 2001.

[2] N. Carlevaris-Bianco and R. M. Eustice. Conservative edge sparsification for graph slam node removal. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 854–860. IEEE, 2014.

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

[4] R. M. Eustice, H. Singh, and J. J. Leonard. Exactly sparse delayed-state filters. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2417–2424. IEEE, 2005.

[5] R. Faragher and R. Harle. Smartslam–an efficient smartphone indoor positioning system exploiting machine learning and opportunistic sensing. In *ION GNSS*, volume 13, pages 1–14, 2013.

[6] U. Frese, P. Larsson, and T. Duckett. A multilevel relaxation algorithm for simultaneous localization and mapping. *Robotics, IEEE Transactions on*, 21(2):196–207, 2005.

[7] C. Gao. *Signal maps for smartphone localisation*. Phd thesis, Computer Laboratory, University of Cambridge, August 2016.

[8] C. Gao and R. Harle. Sequence-based magnetic loop closures for automated signal surveying. In *Indoor Positioning and Indoor Navigation (IPIN), 2015 International Conference on*, pages 1–12. IEEE, 2015.

[9] G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard. A tree parameterization for efficiently computing maximum likelihood maps using gradient descent. In *Robotics: Science and Systems*, 2007.

[10] B. K. Horn. Closed-form solution of absolute orientation using unit quaternions. *JOSA A*, 4(4):629–642, 1987.

[11] J. Huang, D. Millman, M. Quigley, D. Stavens, S. Thrun, and A. Aggarwal. Efficient, generalized indoor wifi graphslam. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1038–1043, May.

[12] J. Jung, T. Oh, and H. Myung. Magnetic field constraints and sequence-based matching for indoor pose graph slam. *Robotics and Autonomous Systems*, 70:92–105, 2015.

[13] M. Kaess, A. Ranganathan, and F. Dellaert. isam: Incremental smoothing and mapping. *Robotics, IEEE Transactions on*, 24(6):1365–1378, 2008.

[14] K. Konolige, G. Grisetti, R. Kümmerle, W. Burgard, B. Limketkai, and R. Vincent. Efficient sparse pose adjustment for 2d mapping. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 22–29. IEEE, 2010.

[15] R. Kümmerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g 2 o: A general framework for graph optimization. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3607–3613. IEEE, 2011.

[16] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous robots*, 4(4):333–349, 1997.

[17] M. Montemerlo and S. Thrun. Large-scale robotic 3-d mapping of urban structures. In *Experimental Robotics IX*, pages 141–150. Springer, 2006.

[18] R. Mur-Artal, J. Montiel, and J. D. Tardós. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[19] E. Olson. *Robust and efficient robotic mapping*. Phd thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2008.

[20] E. Olson, J. Leonard, and S. Teller. Fast iterative alignment of pose graphs with poor initial estimates. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 2262–2269. IEEE, 2006.

[21] S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics*. MIT press, 2005.

[22] S. Thrun, Y. Liu, D. Koller, A. Y. Ng, Z. Ghahramani, and H. Durrant-Whyte. Simultaneous localization and mapping with sparse extended information filters. *The International Journal of Robotics Research*, 23(7-8):693–716, 2004.