SPARSE POSE ADJUSTMENT - APPLICATION OF LEVENBERG-MARQUARDT (LM) NONLINEAR OPTIMIZER

Abstract. Pose graph is a widely known graph based formulation for the Simultaneous Localization and Mapping problem in the field of robotics. Given a sequence of observations for example, odometry data or laser scan data, the aim is to estimate the set of poses of the robot that fits with the observed data. For the final project of 18.337, a pose graph optimization algorithm called Sparse Pose Adjustment (SPA) [1] has been implemented.

1. Introduction. In the pose graph formulation of problem [2], the nodes represent the position of the robot (called poses), and the edges represent the constraints of spatial measurement data. A typical instance of this formulation is a large graph, containing thousands of nodes and edges. In most real world datasets, the connections (edges) in the 2D pose graph are sparse due to the limited range of sensors which only records local measurements. This sparsity pattern can be exploited to efficiently calculate the Jacobian and Hessian of the error function to minimize the cost. [1]

This report is organized as follows. The next section gives a description of problem formulation, followed by Experimental results, comparison with the previous implementations, effect of initialization, insight into the code - emphasizing on efficient and inefficient parts and the conclusion. The most recent code and the report can be found in here. The github repository also mentions how to run the code.

2. Problem Formulation. The problem formulation is similar to as specified in the paper [1]. As a brief overview of: Represent the poses in the graph by $c_i = [x, y, \theta]$, $i \in \{1, 2, ..., n\}$ with n being the number of poses in the graph. Stack these c_i's vertically to obtain the variable $3n$x1 matrix X, which needs to be optimized. Now, start from an initial guess (discussed more in section 5) X and calculate the offset between poses c_i and c_j, for which corresponding edges are present in the graph. The deviation of these offsets from the dataset constraints are used to find the error e_{ij}. The total squared cost for the current estimate X is $\chi^2 = \sum_{i,j} e_{ij}^T \Lambda_{ij} e_{ij}$, where Λ_{ij} is the covariance, obtained from the dataset. This is a non linear problem, solved using LM algorithm, with the linear solver being cholesky with permutation (AMD).

$$\Lambda = \begin{bmatrix} \lambda_{ab} & & \\ & \ddots & \\ & & \lambda_{mn} \end{bmatrix} \tag{2.1}$$

$$J = \frac{\partial \mathbf{e}}{\partial \mathbf{x}} \tag{2.2}$$

The Levenberg-Maquardt equation is set up as

$$(\mathbf{H} + \lambda diag\mathbf{H})\Delta x = \mathbf{J}^T \Lambda \mathbf{e} \tag{2.3}$$

3. Solving LM-equation. The above LM equation can be modeled as a linear equation

$$A\Delta x = b \tag{3.1}$$

Using Cholesky or (PCG) we factorize the A matrix as $P^T L L^T P$. The P represents the permutation of the rows and cholesky with permutation was found to be much

faster as compared to without permutation. Julia has a user friendly implementation of cholesky decomposition from which one can obtain $P^T L$. Lets call $R = P^T L$. Then

$$y = Rb$$

$$\Delta x = R^T y$$

.

4. Exeprimental Results. The code was run on couple of datasets (obtained from [3]) and the corresponding parameters used are as follows.

- **city10000.g2o** 10,000 nodes, 20,687 edges, Initial cost: 5.9e7, Final cost: 1.02e5, Iterations:2, Time: 0.86 seconds, initial $\lambda = 1e-4$, α: 0.8
- **manhattan.g2o** 3500 nodes, 5453 edges, Initial cost: 1.56e6, Final cost: 1.052e5, Iterations:2, Time: 0.259 seconds, initial λ: 1e-7, α: 0.7
- **intel.g2o** 1728 nodes, 2512 edges, Initial cost: 1.41e5, Final cost: 2.73e2, Iterations:1, Time: 0.067 seconds, initial λ: 1e-6, α: 0.9
- **CSAIL.g2o** 1045 nodes, 1171 edges, Initial cost: 5.9e5, Final cost: 2.411e3, Iterations:1, Time: 0.03 seconds, initial λ: 1e-6, α: 0.7

The final poses were obtained as sgown in figure 1.

4.1. Benchmarking. This project's implementation was tested for four datasets, as shown in figure 2. The time plot generated from the execution is shown on the left. The time plot from the paper [1] (which is based on C++ code) is also shown for comparison. Please note the scale used in the figures (figures have different scales for x and y axis). The time taken by our code is of the same order of magnitude.

5. Initializing Pose Matrix. There are $3n$ parameters and this is in the order 10^3, and thus a random initial guess is highly unlikely to work. To intialize the poses, a spanning tree initialization was implemented. The first pose was initialized as [0,0,0] with the subsequent nodes connected to node 1 are initialized, and then their children nodes and so on. An equivalent description of a spanning tree is the initialization in breadth first order.

Another commonly used approach is the odometry initialization in which the nodes are initialized according to their number. The spanning tree works best for SPA because -

The SPA is sensitive to initial pose estimate, as can be seen from the figure 3. The following is the city10000 data set, which has 10,000 poses (nodes) and 20687 constraints (edges).

6. Code Profiling.

6.1. A brief overview of Code Optimization techniques used. Wherever possible, fuse operator has been used to improve efficiency of matrix additions and to reuse the variables (and thus minimizing new allocations). Pointer change has also been used to update the current estimate of poses, and to update the error matrix (this is to reuse the variables). Error vector, which is of the type [Int64, Int64, [δx, δy, $\delta\theta$]], is updated in each iteration, instead of re-initializing a new error vector, and only the last element is updated.

The function arguments have specified type, as the functions are used and are intended to be used only with specific types of arguments. Abstract types have been avoided, and type has been declared in every initialization done in the code.

Bound checking has been prevented by using @inbounds.

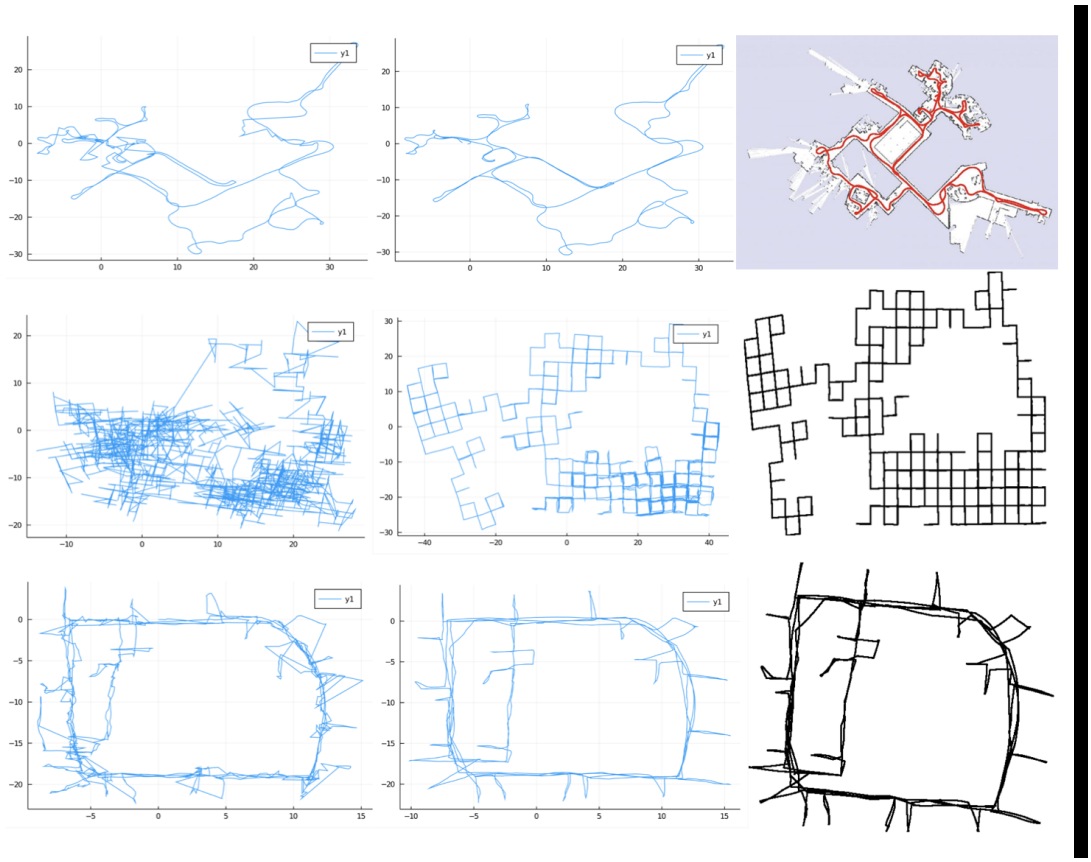

FIG. 1. *Row 1: CSAIL.g2o dataset; Row 2: Manhattan3500.g2o dataset; Row 3: intel.g2o dataset; Column 1: Initial Poses; Column 2: Optimized Poses; Column 3: Actual Poses*

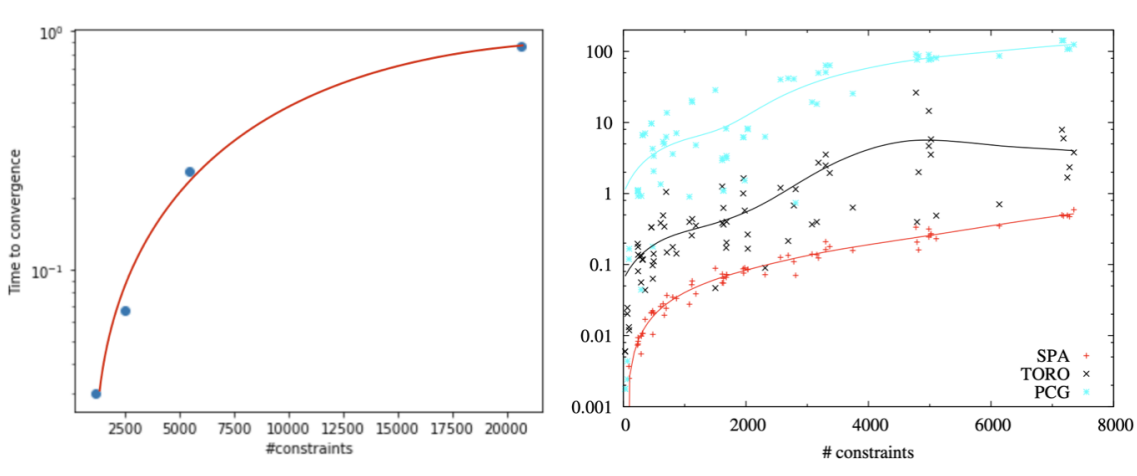

FIG. 2. *Left: Julia's SPA time analysis; Right: Plot taken from [1]*

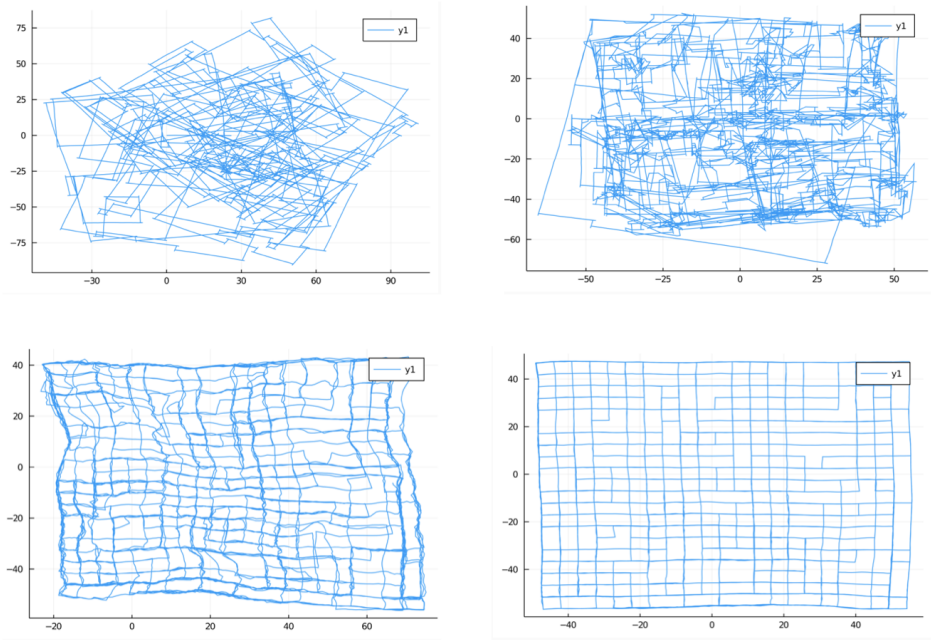

FIG. 3. *Top Left: Odometry Intialized Poses; Top Right: Spanning Tree Intialized Poses; Bottom Left: Optimized Poses for Odometry Initialization; Bottom Right: Optimized Poses for Spanning Tree Initialization*

The Pose array (containing variables to be optimized) has shape 3*numPoses, as it was often required in the code to access pose(i) and since Julia stores matrices in columnwise order, each individual pose is a column.

Function CreateSparse: This function currently takes most percent of total time and following inefficiencies in the current code has been identified.

The Hessian matrix (sparse), is constructed using three jacobians, each of size 3x3, These small jacobians are calculated for every constraint and then added as per the position in which they should appear in the sparse matrix to obtain the final Jacobian. ($|z|$ represents the number of constraints). Thus $2|z|$ Jacobians are allocated in every iteration. Further in each call of this function, roughly $6|Z|$ matrix products are occurring, each giving rise to new matrices as a result of each matrix multiplication.

A likely approach to remove this inefficiency is to merge the part where smaller jacobians are calculated (and accumulated to form "big" Jacobian) and the part where elements from the "big" jacobian ($\frac{\partial \mathbf{e}}{\partial \mathbf{x}}$) are used to form the sparse matrix.

CreateSparse currently uses append!, which is dynamically adding new elements to the array: this can be improved if the number of elements in each column and row in the hessian can be precalculated. This would require some extra code which finds the length of these arrays beforehand and check if this extra computation do not affect the code negatively.

7. Conclusion and Future Work. The Julia implementation of SPA as compared to C++ code which the author of the paper [1] used performed relatively well - Julia code also performed within the same order of magnitude. Exact comparison

was not possible because this project's implementation was tested on only 4 datasets which were easily available. Further, the final cost to which the poses have been optimized also greatly affects the time required to reach that optimal point. Some of the parts of the code have been isolated where large number of allocations are happening and can be improved further. Moreover, some parts of the code can be parallelized, for example, creating error matrices finding "small" jacobians, and so on.

8. Acknowledgement. The Julia documentation for various libraries was very helpful [4]. The datasets for testing were taken from [2].

REFERENCES

[1] K. Konolige, G. Grisetti, R. Ku mmerle, W. Burgard, B. Limketkai, and R. Vincent. Sparse pose adjustment for 2d mapping. In Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS), 2010.

[2] g2o:A General Framework for Graph Optimization (https://www.researchgate.net/publication/224252449_G2o_A_general_framework_for_graph_optimization)

[3] L. Carlone and A. Censi. From Angular Manifolds to the Integer Lattice: Guaranteed Orientation Estimation With Application to Pose Graph Optimization. IEEE Trans. Robotics, 30(2):475-492, 2014.

[4] https://docs.julialang.org/en/v1/ : documentations for various libraries