Pose Graph SLAM

Valerio Sanelli
February 7, 2016

1 INTRODUCTION

The project aim was to implement a SLAM algorithm in order to determine the most likely configuration of a pose graph.
A pose graph is a representation for the SLAM problem in which every node represent a robot pose while the edges represents sensor measurements that constraints them.
Such measurements are affected by noise, so the main objective of the algorithm is to find the configuration of the nodes that is maximally consistent with the measurements.
The solution is found by solving a large error minimization problem.
A general Graph-based SLAM algorithm interleaves two steps, graph construction and optimization, however this project will focus on the latter.

2 SOLUTION

The solution is developed in C++ using the Eigen library for matrix manipulations and the SuiteSparse package for linear system solving. The input for the algorithm is a g2o map representing the graph.
The general methodology ([1]) and the algorithm are listed in the following.

STATE VARIABLES The state variables are 2D robot poses. A pose is a 3x1 vector representing a translation and a rotation with respect to the world reference frame. The extended parameterization for the robot position is a 3x3 transformation matrix with the following form:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{R}(\theta) & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{bmatrix} \tag{2.1}$$

$\boldsymbol{R}(\theta)$ represent the orientation of the robot local frame wrt the z axis of the world reference frame. $\boldsymbol{t}(x,y)$ is the translation of the robot wrt the origin. Is it possible to switch from the minimal to the extended representation simply using a function called v2t and defined as follow:

$$v2t(Vector v) = \begin{bmatrix} cos(v(3)) & -sin(v(3)) & v(1) \\ sin(v(3)) & cos(v(3)) & v(2) \\ 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

where v is a vector representing a robot pose.

PARAMETRIATION OF THE INCREMENTS An increment in a robot pose is a 3x1 vector of the form:

$$\Delta x = (\Delta t_x, \Delta t_x, \theta)$$

The space state is non Euclidean, since is composed by a translation vector and a rotation. However, in this simple 2D case, the singularities introduced by the representation can be easily recovered by normalizing the angles to $[-\pi,\pi)$ after applying the increments. Thus there is no need to define $\boxplus$ operator.

MEASUREMENT FUNCTION The measurement function predicting the measurement $z_{i,j}$ will map the position of the robot j in the frame of robot i as follows:

$$h_{i,j}(x) = X_i^{-1} \cdot X_j$$

ERROR FUNCTION The error function would be:

$$e_{ij} = t2v(Z_{ij}^{-1}(X_i^{-1} \cdot X_j)$$

where tv2 is the converse of v2t previously defined. It takes an homogeneous transformation matrix T as input and computes the corresponding minimal representation v as follow:

v[1] = T[1,3]
v[2] = T[2,3]
v[3] = atan2(T(2,1),T(1,1))

However the error function could be written using rotation matrices and translation vectors to ease the computation of the jacobians later:

$$e_{ij} = Z_{ij}^{-1} \begin{bmatrix} \boldsymbol{R}_i^t(t_j - t_i) \\ \theta_j - \theta_i \end{bmatrix}$$

JACOBIANS Since the error function of a constraint depends only on the values of two nodes, the Jacobian has the following form:

$$J_{ij} = (0 \ldots 0 \, A_{ij} \, 0 \ldots 0 \, B_{ij} \, 0 \ldots 0)$$

A_{ij} and B_{ij} are the derivatives of the error function with respect to x_i and x_j. Using the previously defined error function, the two 3x3 blocks corresponding to A and B would be:

$$\boldsymbol{A}_{ij} = \begin{bmatrix} -R_z^t R_i^t & R_z^t \frac{dR_i^t}{d\theta_i}(\boldsymbol{t}_j - \boldsymbol{t}_i) & \\ 0 & 0 & -1 \end{bmatrix}$$

$$\boldsymbol{B}_{ij} = \begin{bmatrix} R_z^t R_i^t & 0 & \\ 0 & 0 & 1 \end{bmatrix}$$

GAUSS-NEWTON ALGORITHM Once all the needed quantities are defined we can put things together in the Algorithm 1 ([2]):

Algorithm 1 Computes the mean $\mathbf{x}^*$ and the information matrix $\mathbf{H}*$ of the multivariate Gaussian approximation of the robot pose posterior from a graph of constraints.

Require: $\breve{\mathbf{x}} = \breve{\mathbf{x}}_{1:T}$: initial guess. $\mathcal{C} = \{\langle \mathbf{e}_{ij}(\cdot), \mathbf{\Omega}_{ij} \rangle\}$: constraints

Ensure: $\mathbf{x}^*$: new solution, $\mathbf{H}^*$ new information matrix

```
// find the maximum likelihood solution
while ¬converged do
  b ← 0      H ← 0
  for all ⟨e_ij, Ω_ij⟩ ∈ C do
    // Compute the Jacobians A_ij and B_ij of the error function
    A_ij ← ∂e_ij(x)/∂x_i |_{x=x̆}      B_ij ← ∂e_ij(x)/∂x_j |_{x=x̆}
    // compute the contribution of this constraint to the linear system
    H_[ii] += A_ij^T Ω_ij A_ij      H_[ij] += A_ij^T Ω_ij B_ij
    H_[ji] += B_ij^T Ω_ij A_ij      H_[jj] += B_ij^T Ω_ij B_ij
    // compute the coefficient vector
    b_[i] += A_ij^T Ω_ij e_ij      b_[j] += B_ij^T Ω_ij e_ij
  end for
  // keep the first node fixed
  H_[11] += I
  // solve the linear system using sparse Cholesky factorization
  Δx ← solve(H Δx = −b)
  // update the parameters
  x̆ += Δx
end while
x* ← x̆
H* ← H
// release the first node
H*_[11] −= I
return ⟨x*, H*⟩
```

3 IMPLEMENTATION

The code was developed completely in C++ and consist of just one file. I will enlist the data structures and the function used in the following.

STRUCTURES there are some global defined variables that represents the various quantities needed by the algorithm. In particular, for the graph representation:

- **vmeans**: it's a $3 \times n$ matrix, where n is the number of nodes. Each column represent a node as a 2D pose.
- **emeans**: it's a $3 \times m$ matrix, where m is the number of edges. Each column represent an edge, that is an odometry measurement.
- **einfs**: it's a vector composed by 3×3 matrices. The matrix in position k is the information matrix of the *k-th* edge.
- **eids**: it's a $2 \times m$ matrix. The column k represent the ids of the nodes involved in the *k-th* constraint as $\begin{bmatrix} id_i \\ id_j \end{bmatrix}$.

The matrices are represented using the standard *MatrixXd* class from Eigen. For the linearized system we have:

- **H**: the Hessian of the system, with dimension $(n \times 3) \times (n \times 3)$. This is represented efficiently using the *SparseMatrix<T>* module of Eigen.
- **b**: is the coefficient vector and has dimension $(n \times 3) \times 1$. It is represented using the *VectorXd* class from Eigen.
- **A**: it's the 3×3 block representing the jacobian of the error function wrt x_i. It's a *MatrixXd*.
- **B**: it's the 3×3 block representing the jacobian of the error function wrt x_j. It's a *MatrixXd*.

FUNCTIONS

- **read_graph(const string vfile,const string efile)**: read vertices/edges from the two files specified as input and initialize the matrices representing the graph.
- **write_graph()**: write the optimized graph in a g2o file in order to visualize it using a Matlab script (showG2OFiles.m).
- **main()**: is the principal function. It is responsible for reading and writing the graph and iterates over the function that solves the linear system.
- **linearize_and_solve()**: is the core of the algorithm. Linearizes and solves one time the least square slam problem specified by vmeans, eids, emeans and einfs. Returns a new solution computed from the initial guess in vmeans.

- **linear_factors(int k)**: computes the Taylor expansion of the error function of the *k-th* edge. Returns the error, A and B jacobians.
- **v2t(Vector3d v)**: this function computes a homegeneous transformation matrix from the given pose vector.
- **linear_factors(int k)**: this function computes a pose vector from the given homegeneous transformation matrix.

  I also defined a bunch of functions only for debugging purpose: printB() and printHessian().

4 Results and Discussion

5 References

[1] G.Grisetti, "Notes on Least-Squares Squares and SLAM".
[2] G. Grisetti, R. Kummerle, C. Stachniss, W. Burgard, "A Tutorial on Graph-Based SLAM".