1 SLAM with LIDAR Measurements

Motivation

1.1 LIDAR Sensors

TBD See slides.

1.2 Localization via ICP

TBD See slides.

1.3 PoseSLAM

SLAM is **Simultaneous Localization and Mapping**. In the SLAM problem the goal is to localize a robot using the information coming from the robot's sensors. The additional wrinkle in SLAM is that we do *not* know the map a priori, and hence we have to infer the unknown map simultaneously with localization with respect to the evolving map.

PoseSLAM is a variant of SLAM that uses pose constraints as the basic building block, and where we optimize over the unknown vehicles poses. We do not explicitly optimize over a map: that is reconstructed after the fact.

To represent the pose of a vehicle, recall that 2D poses $T \doteq (x, y, \theta)$ form the Special Euclidean group $SE(2)$, and can be represented by 3×3 matrix of the form

$$T = \left[\begin{array}{cc|c} \cos\theta & -\sin\theta & x \\ sin\theta & \cos\theta & y \\ \hline 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{cc} R & t \\ 0 & 1 \end{array}\right]$$

with the 2×1 vector t representing the position of the vehicle, and R the 2×2 rotation matrix representing the vehicle's orientation in the plane.

Note that this representation generalizes equally to three dimensions, but of course t will be a three-vector, and R will be a 3×3 rotation matrix representing the 3DOF attitude of the vehicle. The latter can be decomposed into roll, pitch, and yaw, if so desired.

The PoseSLAM problem is then:

> given a set of noisy relative measurements or **pose constraints** $\tilde{T}_{ij}$, recover the optimal set of poses T_i^* that maximizes the posteriori probability, i.e., recover the MAP solution.

In the case of mapping for autonomous driving, these relative measurements can be derived from performing ICP between overlapping scans. We can use GPS and/or IMU measurements to decide which scans overlap, so that we do not have to compare $O(n^2)$ scans. Depending on the situation, we can optimize for 3D or 2D poses, in the way we will discus below. Afterwards, we can reconstruct a detailed map by transforming the local LIDAR scans into the world frame, using the optimized poses T_i^*.

1.4 The PoseSLAM Factor Graph

In our factor-graph-based view of the world, a pose constraint is represented as a factor. As before, the factor graph represent the posterior distribution over the unknown pose variables $\mathcal{T} = \{T_1 \ldots T_5\}$ given the known measurements:

$$\phi(\mathcal{T}) = \prod_i \phi_i(\mathcal{T}_i). \tag{1}$$

The factor graph encodes which factors are connected to which variables, exposing the sparsity pattern of the corresponding estimation problem.

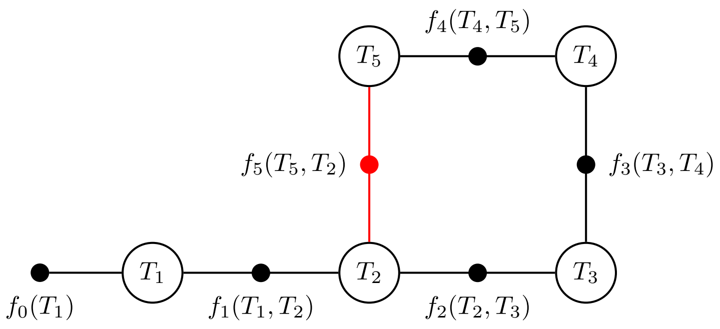

Figure 1: PoseSLAM factor graph example.

An example is shown in Figure 1. The example represents a vehicle driving around, and taking LIDAR scans at 5 different world poses, represented by T_1 to T_5. The factors f_1 to f_4 are binary factors representing the pose constraints obtained by matching successive LIDAR scans. The factor $f_5(T_5, T_2)$ is a so-called "loop closure" constraint: rather than derived from two successive scans, this one is derived from matching the scan taken at T_5 with the one at T_2. Detecting such loops can be done through a variety of means. The final, unary factor $f_0(T_1)$ is there to "anchor" the solution to the origin: if it is not there, the solution will be undetermined. Another way to anchor the solution is to add unary factors at every time-step, derived from GPS.

Finding the MAP in the case that variables are continuous and measurements are linear combinations of them can be done via least-squares. Above we have discussed MAP inference for discrete variables, and we have discussed probability distributions for continuous variables, but we have never put the two together. In the case of measurements corrupted by zero-mean Gaussian noise, we can recover the MAP solution by minimization. Recall that a multivariate Gaussian density **with mean** μ and **variance** σ^2 is given by

$$\mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right\}. \tag{2}$$

If we focus our attention in PoseSLAM on just the x coordinates, then we predict relative measurements $\tilde{x}_{ij}$ by

$$\tilde{x}_{ij} \approx h(x_i, x_j) = x_j - x_i$$

and each factor in Figure 1 could be written as

$$\phi(x_i, x_j) = \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(x_j - x_i - \tilde{x}_{ij}\right)^2\right\}, \tag{3}$$

where we assumed $\sigma = 1$ for now. By taking the negative log, maximizing the posterior corresponds to minimizing the following sum of squares:

$$\mathcal{X}^* = \arg\min_{\mathcal{T}} \sum_k \frac{1}{2}\left(x_j - x_i - \tilde{x}_{ij}\right)^2$$

Linear least squares problems like these are easily solved by numerical computing packages like MATLAB or numpy.

Unfortunately, in the PoseSLAM case we cannot use linear least squares, because poses are not simply vectors, and the measurements are not simply linear functions of the poses. Indeed, in PoseSLAM both the prediction $h(T_i, T_j)$ and the measurement $\tilde{T}_{ij}$ are relative poses. The measurement prediction function $h(.)$ is given by

$$h(T_i, T_j) = T_i^{-1} T_j$$

and the measurement error to be minimized is

$$\frac{1}{2} \left\| \log \left(\tilde{T}_{ij}^{-1} T_i^{-1} T_j \right) \right\|^2 \tag{4}$$

where $\log : SE(2) \to \mathbb{R}^3$ denotes a map from $SE(2)$ to a three-dimensional local coordinate vector ξ, which will be defined in detail below.

1.5 Nonlinear Optimization for PoseSLAM

There are two ways out of the nonlinear quandary. The first is to realize that the only non-linearities stem from the sin and cos terms in the poses, associated with the unknown orientations θ_i. Hence, one solution is to try and solve for the orientations first, and then solve for the translations using linear least squares, exactly as above. This approach is known as **rotation averaging** followed by linear translation recovery. Below we will take a second route, which is to use **nonlinear optimization**, discussed below.

As discussed, the error expressions (4) are *nonlinear*, and we cannot directly optimize over the poses T_i. Instead, we will locally linearize the problem and solve the corresponding linear problem using least-squares, and iterate this until convergence. We do this by, at each iteration, parameterizing a pose T by

$$T \approx \bar{T} \Delta(\xi) \tag{5}$$

where ξ are 3D local coordinates $\xi \doteq (\delta x, \delta y, \delta \theta)$ and the incremental pose $\Delta(\xi) \in SE(2)$ is defined as

$$\Delta(\xi) = \left[\begin{array}{cc|c} 1 & -\delta\theta & \delta x \\ \delta\theta & 1 & \delta y \\ \hline 0 & 0 & 1 \end{array} \right]$$

which you can recognize as a small angle approximation of . In 3D the local coordinates ξ is 6-dimensional, and the small angle approximation is defined as

$$\Delta(\xi) = \left[\begin{array}{ccc|c} 1 & -\delta\theta_z & \delta\theta_y & \delta x \\ \delta\theta_z & 1 & -\delta\theta_x & \delta y \\ -\delta\theta_y & \delta\theta_x & 1 & \delta z \\ \hline 0 & 0 & 0 & 1 \end{array} \right]$$

With this new notation, we can approximate the nonlinear error (4) by a linear approximation:

$$\frac{1}{2} \left\| \log \left(\tilde{T}_{ij}^{-1} T_i^{-1} T_j \right) \right\|^2 \approx \frac{1}{2} \left\| A_i \xi_i + A_j \xi_j - b \right\|^2. \tag{6}$$

For $SE(2)$ the matrices A_i and A_j are the 3×3 **or Jacobian matrices** and b is a 3×1 bias term. The above provides a linear approximation of the term within the norm as a function of the incremental local coordinates ξ_i and ξ_j. Deriving the detailed expressions for these Jacobians is beyond the scope of this document, but suffice to say that they exist and not too expensive to compute. In three dimensions, the Jacobian matrices are 6×6 and 16×6, respectively.

The final optimization will—in each iteration—minimize over the local coordinates of all poses by summing over all pose constraints. If we index those constraints by k, we have the following least squares problem:

$$\Xi^* = \arg\min_{\Xi} \sum_k \frac{1}{2} \left\| A_{ki}\xi_i + A_{kj}\xi_j - b_k \right\|^2 \tag{7}$$

where $\Xi \doteq \{\xi_i\}$, the set of all incremental pose coordinates.

After solving for the incremental updates Ξ, we update all poses using equation 5 and check for convergence. If the error does not decrease significantly we terminate, otherwise we linearize and solve again, until the error converges. While this is not guaranteed to converge to a global minimum, in practice it does so if there are enough relative measurements and a good initial estimate is available. For example, GPS can provide us with a good initial estimate. However, especially in urban environments GPS can be quite noisy, and it could happen that the map quality suffers by converging to a bad local minimum. Hence, a good quality control process is absolutely necessary in production environments.

For SLAM we typically use specializes packages such as G2O, Ceres, or GTSAM that exploit the sparsity of the factor graphs to dramatically speed up computation. Note that MATLAB and/or numpy can solve sparse least squares problems: the specialized SLAM packages simply provide the translation as well as the calculation of the Jacobian matrices above.

In summary, the algorithm for nonlinear optimization is

- Start with an initial estimate $\mathcal{T}^0$
- Iterate:
    1. Linearize the factors $\frac{1}{2}\left\|\log\left(\tilde{T}_{ij}^{-1}T_i^{-1}T_j\right)\right\|^2 \approx \frac{1}{2}\left\|A_i\xi_i + A_j\xi_j - b\right\|^2$
    2. Solve the least squares problem $\Xi^* = \arg\min_{\Xi} \sum_k \frac{1}{2}\left\|A_{ki}\xi_i + A_{kj}\xi_j - b_k\right\|^2$
    3. Update $T_i^{t+1} \leftarrow T_j^t \Delta(\xi_i)$
- Until the nonlinear error $J(\mathcal{T}) \doteq \sum_k \frac{1}{2}\left\|\log\left(\tilde{T}_{ij}^{-1}T_i^{-1}T_j\right)\right\|^2$ converges.

1.6 Optimization with GTSAM

The GTSAM toolbox (GTSAM stands for "Georgia Tech Smoothing and Mapping") toolbox is a BSD-licensed C++ library based on factor graphs, developed at the Georgia Institute of Technology by myself, many of my students, and collaborators. It provides state of the art solutions to the SLAM and SFM problems, but can also be used to model and solve both simpler and more complex estimation problems. More information is available at `http://gtsam.org`.

GTSAM exploits sparsity to be computationally efficient. Typically measurements only provide information on the relationship between a handful of variables, and hence the resulting factor graph will be sparsely connected. This is exploited by the algorithms implemented in GTSAM to reduce computational complexity. Even when graphs are too dense to be handled efficiently by direct methods, GTSAM provides iterative methods that are quite efficient regardless.

The following C++ code, included in GTSAM as an example, creates the factor graph from Figure 1 in code:

```
1  NonlinearFactorGraph graph;
2  auto priorNoise = noiseModel::Diagonal::Sigmas((Vector(3)<< 0.3, 0.3, 0.1));
3  graph.add(PriorFactor<Pose2>(1, Pose2(0,0,0), priorNoise));
4  
5  // Add odometry factors
6  auto model = noiseModel::Diagonal::Sigmas((Vector(3)<< 0.2, 0.2, 0.1));
7  graph.add(BetweenFactor<Pose2>(1, 2, Pose2(2, 0, 0     ), model));
8  graph.add(BetweenFactor<Pose2>(2, 3, Pose2(2, 0, M_PI_2), model));
9  graph.add(BetweenFactor<Pose2>(3, 4, Pose2(2, 0, M_PI_2), model));
10 graph.add(BetweenFactor<Pose2>(4, 5, Pose2(2, 0, M_PI_2), model));
11 
12 // Add pose constraint
13 graph.add(BetweenFactor<Pose2>(5, 2, Pose2(2, 0, M_PI_2), model));
```

Listing 1: Building a graph in C++

Lines 1-4 create a nonlinear factor graph and add the unary factor $f_0(T_1)$. As the vehicle travels through the world, it creates binary factors $f_t(T_t, T_{t+1})$ corresponding to odometry, added to the graph in lines 6-12 (Note that M_PI_2 refers to pi/2). But line 15 models a different event: a **loop closure**. For example, the vehicle might recognize the same location using vision or a laser range finder, and calculate the geometric pose constraint to when it first visited this location. This is illustrated for poses T_5 and T_2, and generates the (red) loop closing factor $f_5(T_5, T_2)$.

1.7 Using the python Interface

GTSAM It also provides both a MATLAB and a python interface which allows for rapid prototype development, visualization, and user interaction. The python library can be imported directly into a Google colab via "import gtsam". A large subset of the GTSAM functionality can be accessed through wrapped classes from within python . The following code excerpt is the python equivalent of the C++ code in Listing 1:

```
1  graph = gtsam.NonlinearFactorGraph()
2  priorNoise = gtsam.noiseModel_Diagonal.Sigmas(vector3(0.3, 0.3, 0.1))
3  graph.add(gtsam.PriorFactorPose2(1, gtsam.Pose2(0, 0, 0), priorNoise))
4  
5  # Create odometry (Between) factors between consecutive poses
6  model = gtsam.noiseModel_Diagonal.Sigmas(vector3(0.2, 0.2, 0.1))
7  graph.add(gtsam.BetweenFactorPose2(1, 2, gtsam.Pose2(2, 0, 0), model))
8  graph.add(gtsam.BetweenFactorPose2(2, 3, gtsam.Pose2(2, 0, pi/2), model))
9  graph.add(gtsam.BetweenFactorPose2(3, 4, gtsam.Pose2(2, 0, pi/2), model))
10 graph.add(gtsam.BetweenFactorPose2(4, 5, gtsam.Pose2(2, 0, pi/2), model))
11 
12 # Add the loop closure constraint
13 graph.add(gtsam.BetweenFactorPose2(5, 2, gtsam.Pose2(2, 0, pi/2), model))
```

Listing 2: Building a graph in python

Note that the code is almost identical, although there are a few syntax and naming differences:

- Objects are created by calling a constructor instead of allocating them on the heap.

- ***Vector*** and ***Matrix*** classes in C++ are just numpy arrays in python.
- As templated classes do not exist in python, these have been hardcoded in the wrapper, e.g., ***PriorFactorPose2*** corresponds to the C++ class ***PriorFactor<Pose2>***, etc.

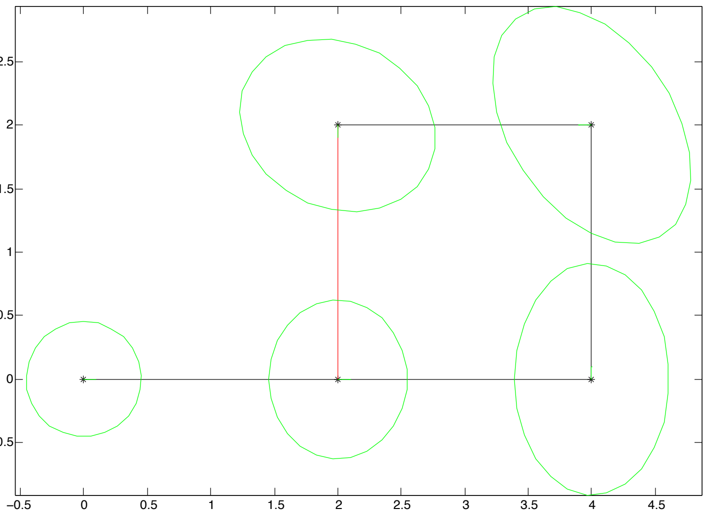

Figure 2: The result of running optimize on the factor graph in Figure 1.

We can optimize this factor graph, by creating an initial estimate of type ***Values***, and creating and running an optimizer. This is illustrated in the listing below:

```python
# Create the initial estimate
initial_estimate = gtsam.Values()
initial_estimate.insert(1, gtsam.Pose2(0.5, 0.0, 0.2))
initial_estimate.insert(2, gtsam.Pose2(2.3, 0.1, -0.2))
initial_estimate.insert(3, gtsam.Pose2(4.1, 0.1, pi/2))
initial_estimate.insert(4, gtsam.Pose2(4.0, 2.0, pi))
initial_estimate.insert(5, gtsam.Pose2(2.1, 2.1, -pi/2))

# Optimize the initial values using a Gauss-Newton nonlinear optimizer
optimizer = gtsam.GaussNewtonOptimizer(graph, initial_estimate)
result = optimizer.optimize()
print("Final Result:\n{}".format(result))
```

Listing 3: Optimizing

The result is shown graphically in Figure 2, along with covariance ellipses shown in green. These covariance ellipses in 2D indicate the marginal over position, over all possible orientations, and show the area which contain 68.26% of the probability mass (in 1D this would correspond to one standard deviation). The graph shows in a clear manner that the uncertainty on pose T_5 is now much less than if there would be only odometry measurements. The pose with the highest uncertainty, T_4, is the one furthest away from the unary constraint $f_0(T_1)$, which is the only factor tying the graph to a global coordinate frame.

The figure above was created using an interface that allows you to use GTSAM from within MATLAB, which provides for visualization and rapid development. However, below we will focus on the python interface, which is also available.

Summary

We briefly summarize what we learned in this section:

1. LIDAR is a key sensor for autonomous driving
2. Localization can be done with LIDAR, or image-based
3. PoseSLAM: a SLAM variant using ICP pose constraints
4. The PoseSLAM factor graph graphically shows the constraints
5. MAP/MAP solution can be done via nonlinear optimization