README for FastSLAM

Kirill Kouzoubov

1 Overview

This document describes implementation of FastSLAM algorithm for point-landmarks.

2 Data Flow and Data Structure

First we obtain raw Laser and Odometry data using *OdoSource* and *LaserSource* classes.

Odometry data is then fed into the *SLAM* class which passes it on to underlying *ParticleFilter* which in turn passes it on to it's *MotionModel*. *ParticleFilter* then samples control inputs from *MotionModel* to advance particles.

Laser Data is passed to shiny feature extractor function, which extract point landmarks using intensity information. Point landmarks are then added to global observation store (class *ObservationStore*).

Once features have been detected an update is called in *SLAM* class, this triggers evaluation in the underlying *ParticleFilter* that passes on all the information to particles (*SLAMParticle*). Update takes: reference to ObservationStore, indices of new observations, and a TimeStamp of observation.

SLAMParticle then computes predicted robot pose at the time of the observation, adds it to it's path (*OdometryStore*), then passes on the observations and robot pose to the *MapBuilder*.

MapBuilder computes global coordinates of observations, finds subset of the current map that should be visible from the robot pose, runs nearest neighbour on this subset, updates map elements and adds new ones as required, also computes importance weights as it goes. Most of these tasks are actually performed by the *Map* class, which also implements a binary-tree like structure for storing landmarks with branches shared by several trees. This structure reduces memory complexity and copying time when particles are resampled.

Every landmark in the map (*MapEntry*) maintains a list of observations. For every observation it stores: index in the global observation store, and pointer to the robot pose. This information is used to recompute probability of the landmark after it is updated. Probability of the landmark is just a sum of mahalanobis distances squared from all observations to the landmarks location estimate.

SLAMParticle computes it's importance weight as following:

$$\begin{aligned}
w'_k &= w_{k-1} + p(Obs|Map_{k-1})p(Map_{k-1}), && (1)\\
p(Obs|Map_{k-1}) &= exp(-md2(Obs, Map_{k-1})), && (2)\\
p(Map_{k-1}) &= exp(-\sum md2(obs_i, Map_{k-1})), && (3)\\
w_k &= nomalise(w'_k). && (4)
\end{aligned}$$

In here $md2$ is a mahalonobis distance squared function.

3 Source Files

FastSLAM implementation consists of the following files

DataSource.cc– contains classes for accessing odometry and laser data.

DataStore.cc – Storage classes:

OdometryStore – a link list of robot poses, with fancy copying capabilities for sharing common parts of the path for different particles.

ObservationStore – a vector of all observations in robot coordinate frame.

geometry.cc – Defines data structures like Point, PointLandmark, RobotPose, does some math on covariance matrices.

kalman2.cc – Kalman filter for updating pointlandmark position estimate.

matrix2.h – Math on 2x2 matrices, used by kalman filter.

matrix2.cc– test file for matrix2.h

random.cc – Uniform and Gaussian random number generators.

odoModel.cc – Odometry model for the holonomic robot, derives control commands from odometry, provides sampling functions for control commands, translates control commands to odometry.

pfilter.cc – Particle filter implementation, a generic one, should be able to use the same framework for localisation purposes as well.

bisearch.cc – Binary search for particle filter resampling step.

map.cc – Map building. Builds maps of PointLandmarks, has fancy structure for sharing common sub-maps between particles, performs nearest neighbour data association using mahalanobis distances, provides weights for SLAM.

slam.cc – Specialised particle filter for doing SLAM. Defines *SLAMParticle* that implements *Particle* interface. Every *SLAMParticle* has *MapBuilder* that does all the work of map building, also every particle stores it's path, using *OdometryStore* class.

memstat.cc – Debugging module that keeps track of how much storage is used up by maps and odometry, can be turned of by undefining USE_MEMSTAT (see Makefile).

test_pfilter.cc – test file for the particle filter implementation, might be outdate.

odo_test.cc – test file for odometry model.

slam_app.cc– Application that does slam using shiny features, also contains shiny feature extractor function.

4 Compile, Run, etc.

To compile type **make**.

To run type **./slam_app odo_file laser_file output_prefix numParticles**

Example: **./slam_app odo.log las.log slam_out/dat 100**.

Program generates a lot of files, two per particle plus more. For every particle we dump map and odometry in separate files. There is global matlab file **prefix**_mat.m that contains final weight of particles, all observations extracted in robot-centered coordinates and odometry. There are also some debug files generated by the particle filter.

Matlab directory contains matlab files for displaying the results, see README file in that directory for typical usage.