Full Body Planner Doc

Victor Hwang

July 2014

Contents

2.1 EnvInterfaces and co

EnvInterfaces is the overall layer that glues together ROS and the planning environment. This is meant to separate ROS service, topic subscriptions, and IO from the logic. If you want to add callbacks or services, add them here to have access to the proper objects.

This layer handles the subscribers for:

1. planning request callback (this is where the SBPL/OMPL planner is called as well)
2. 2D navmap callback (for base heuristic)
3. 3D octomap callback (CollisionSpaceInterface)
4. Writing stats to file

2.2 Environment

This is where the magic happens. This class inherits from SBPL environment.h, so it's got all the standard stuff (GetSuccs, GetHeuristic, etc). This also hold all the other support objects.

2.3 OccupancyGridUser

This is a dumb name, but this directly handles occupancy grid usage. This is a class with a static member variable, and is inherited by any class that needs knowledge of the grid resolution.

2.4 CollisionSpaceMgr

This handles everything related to the collision space, collision checking, and attached objects. There's a little bit of code here for attaching objects, but there is no good API for it yet.

2.5 ParameterCatalog

This handles grabbing all parameters from files or ROS param server. Each module in the environment has its own struct inside the ParameterCatalog. Typically, the environment will instantiate the module with the relevant catalog.

2.6 PathPostProcessor

This handles reconstructing the path from a vector of state ids and shortcutting.

2.7 Motion Primitive Manager

This handles instantiating the motion primitives. This also loads the correct set of motion primitives when the planning request is received (to allow for different modes of operation).

2.8 HeuristicManager

This was originally made for MHA*, but contains a bunch of stuff for designing a lot of heuristics and instantiating them.

3 State Representation

The highest level representation is the GraphState. This holds member variables for $x, y, z, roll, pitch, yaw$ of the object, the left and right free angles of the arm, and x, y, z, θ of the base. The GraphState also contains a RobotState, which is always the full 18DOF configuration of the robot. In general, we expect the GraphState and RobotState to match, but in the case of the lazy planner, this is not always the case.

The RobotState is made up of a ContBaseState, LeftContArmState, and RightContArmState. The object state is automatically computed from the dominant arm (almost always right arm, unless you're using only the left arm). The RobotState stores everything as continuous values, so no need to worry about discretization problems (unless you convert to discrete and shove them back in).

4 Motion Primitives

Applying the motion primitives can be seen in the **Environment.cpp** file. The MotionPrimitiveMgr maintains a set of active motion primitives based on what the planning request is (dual arm, [left/right] single arm, mobile manipulation [dual/left/right], or navigation).

Each different motion primitive type contains an **apply** method that takes in a graph state and does whatever the motion primitive is set to do. It spits out a successor state (while doing no collision checking), along with a piece of TransitionData, which contains all the intermediate points of the motion. This is fed into the collision checker to handle if the successor is valid.

5 Path Reconstruction

6 Designing heuristics

7 How the lazy functionality is implemented

There's one thing about lazy that makes the code a little messy - because we're lazily generating the successors, a mismatch occurs between the graph state and the stored robotstate. This (supposedly) gets resolved in GetTrueCost.

It can get hairy if things don't get updated correctly, leading to states that don't exist in the hash manager. I've got some functions that can synchronize these two things in GraphState.cpp.

8 How the e-graph functionality is implemented