ROS By Example

R. Patrick Goebel

8. Navigation, Path Planning and SLAM

Now that we have covered the basics of how to control a differential drive robot, we are ready to try one of the more powerful features in ROS; namely, Simultaneous Localization and Mapping or SLAM.

A SLAM-capable robot can build a map of an unknown environment while simultaneously locating itself in that map. Until recently, about the only way to do reliable SLAM was to use a fairly expensive laser scanner to collect the data. With the arrival of the Microsoft Kinect and Asus Xtion cameras, one can now do more affordable SLAM by using the 3D point cloud from the camera to generate a "fake" laser scan. (See the `depthimage_to_laserscan` and `kinect_2d_scanner` packages for two ways to do this.) The TurtleBot is configured to do this out of the box. If you own a TurtleBot, you might want to skip directly to the TurtleBot SLAM tutorial on the ROS Wiki.

Another affordable SLAM robot is the Neato XV-11 vacuum cleaner which includes a 360-degree laser scanner. In fact, you can run the complete Navigation Stack using the XV-11 thanks to the `neato_robot` ROS stack by Michael Ferguson.

In this chapter, we will cover the three essential ROS packages that make up the core of the Navigation Stack:

- `move_base` for moving the robot to a goal pose within a given reference frame
- `gmapping` for creating a map from laser scan data (or simulated laser data from a depth camera)
- `amcl` for localization using an existing map

When we are finished, we will be able to command the robot to go to any location or series of locations within the map, all while avoiding obstacles. Before going further, it is highly recommended that the reader check out the Navigation Robot Setup tutorial on the ROS Wiki. This tutorial provides an excellent overview of the ROS navigation stack. For an even better understanding, check out all of the Navigation Tutorials. And for a superb introduction to the mathematics underlying SLAM, check out Sebastian Thrun's online Artificial Intelligence course on Udacity.

8.1 Path Planning and Obstacle Avoidance using `move_base`

In the previous chapter, we wrote a script to move the robot in a square. In that script, we monitored the `tf` transform between the `/odom` frame and the `/base_link` (or `/base_footprint`) frame to keep track of the distance traveled and the angles rotated. ROS provides a much more elegant way to do the same thing using the `move_base`

package. (Please see the `move_base` Wiki page for a full explanation including an excellent diagram of the various components of the package.)

The `move_base` package implements a ROS action for reaching a given navigation goal. You should be familiar with the basics of ROS actions from the actionlib tutorials on the ROS Wiki. Recall that actions provide feedback as progress is made toward the goal. This means we no longer have to query the odometry topic ourselves to find out if we have arrived at the desired location.

The `move_base` package incorporates the `base_local_planner` that combines odometry data with both global and local cost maps when selecting a path for the robot to follow. The global plan or path is computed before the robot starts moving toward the next destination and takes into account known obstacles or areas marked as "unknown". To actually move the robot, the local planner monitors incoming sensor data and chooses appropriate linear and angular velocities for the robot to traverse the current segment of the global path. How these local path segments are implemented over time is highly configurable as we shall see.

8.1.1 Specifying Navigation Goals Using `move_base`

To specify a navigation goal using `move_base`, we provide a target pose (position and orientation) of the robot with respect to a particular frame of reference. The `move_base` package uses the `MoveBaseActionGoal` message type for specifying the goal. To see the definition of this message type, run the command:

```
$ rosmsg show MoveBaseActionGoal
```

which should produce the following output:

```
Header header
  uint32 seq
  time stamp
  string frame_id
actionlib_msgs/GoalID goal_id
  time stamp
  string id
move_base_msgs/MoveBaseGoal goal
  geometry_msgs/PoseStamped target_pose
    Header header
      uint32 seq
      time stamp
      string frame_id
    geometry_msgs/Pose pose
      geometry_msgs/Point position
        float64 x
        float64 y
        float64 z
      geometry_msgs/Quaternion orientation
        float64 x
        float64 y
        float64 z
        float64 w
```

As you can see, the goal consists of a standard ROS `header` including a `frame_id`, a `goal_id`, and the `goal` itself which is given as a `PoseStamped` message. The `PoseStamped` message type in turn consists of a `header` and a `pose` which itself consists of a `position` and an `orientation`.

While the `MoveBaseActionGoal` message might seem a little complicated when written out in full, in practice we will only have to set a few of the fields when specifying `move_base` goals.

8.1.2 Configuration Parameters for Path Planning

The `move_base` node requires four configuration files before it can be run. These files define a number of parameters related to the cost of running into obstacles, the radius of the robot, how far into the future the path planner should look, how fast we want the robot to move and so on. The four configuration files can be found in the `config` subdirectory of the `rbx1_nav` package and are called:

- `base_local_planner_params.yaml`
- `costmap_common_params.yaml`
- `global_costmap_params.yaml`
- `local_costmap_params.yaml`

We will describe just a few of the parameters here; in particular, those that you will be most likely to tweak for your own robot. To learn about all of the parameters, please refer to the Navigation Robot Setup on the ROS Wiki as well as the parameters section of the `costmap_2d` and `base_local_planner` Wiki pages.

8.1.2.1 base_local_planner_params.yaml

The values listed here are taken from `base_local_planner_params.yaml` in the `rbx1_nav/config/turtlebot` directory and have been found to work fairly well for the TurtleBot and Pi Robot. (Values that work well in the ArbotiX simulator can be found in the directory `rbx1_nav/config/fake`.)

- `controller_frequency: 3.0` – How many times per second should we update the planning process? Setting this value too high can overload a underpowered CPU. A value of 3 to 5 seems to work fairly well for a typical laptop.
- `max_vel_x: 0.3` – The maximum linear velocity of the robot in meters per second. A value of 0.5 is rather fast for an indoor robot so a value of 0.3 is chosen to be conservative.
- `min_vel_x: 0.05`: – The minimum linear velocity of the robot.

- `max_vel_theta: 1.0` – The maximum rotational velocity of the robot in radians per second. Don't set this too high or the robot will overshoot its goal orientation.
- `min_vel_theta: -1.0` – The minimum rotational velocity of the robot in radians per second.
- `min_in_place_vel_theta: 0.5` – The minimum in-place rotational velocity of the robot in radians per second.

- `escape_vel: -0.1` – Speed used for driving during escapes in meters per sec. Note that it must be negative in order for the robot to actually reverse.
- `acc_lim_x: 2.5` – The maximum linear acceleration in the x direction in m/s^2
- `acc_lim_y: 0.0` – The maximum linear acceleration in the y direction in m/s^2. We set this to 0 for a differential drive (non-holonomic) robot since such a robot can only move linearly in the `x` direction (as well as rotate).
- `acc_lim_theta: 3.2` – The maximum angular acceleration in rad/s^2.
- `holonomic_robot: false` – Unless you have an omni-directional drive robot, this is always set to false.

- `yaw_goal_tolerance: 0.1` – How close to the goal orientation (in radians) do we have to get? Setting this value to small may cause the robot to oscillate near the goal.
- `xy_goal_tolerance: 0.1` – How close (in meters) do we have to get to the goal? If you set the tolerance too small, your robot may try endlessly to make small adjustments around the goal location. **NOTE**: Do not set the tolerance less than the resolution of your map (described in a later section) or your robot will end up spinning in place indefinitely without reaching the goal.
- `pdist_scale: 0.8` – The relative importance of sticking to the global path as opposed to getting to the goal. Set this parameter larger than the `gdist_scale` to favor following the global path more closely.
- `gdist_scale: 0.4` – The relative importance of getting to the goal rather than sticking to the global path. Set this parameter larger than the `pdist_scale` to favor getting to the goal by whatever path necessary.
- `occdist_scale: 0.1` – The relative importance of avoiding obstacles.
- `sim_time: 1.0` – How many seconds into the future should the planner look? This parameter together with the next (`dwa`) can greatly affect the path taken by your robot toward a goal.

- `dwa: true` – Whether or not to use the Dynamic Window Approach when simulating trajectories into the future. See the Base Local Planner Overview for more details.

8.1.2.2 `costmap_common_params.yaml`

There are only two parameters in this file that you might have to tweak right away for your own robot:

- `robot_radius: 0.165` – For a circular robot, the radius of your robot in meters. For a non-circular robot, you can use the `footprint` parameter instead as shown next. The value here is the radius of the original TurtleBot in meters.
- `footprint: [[x0, y0], [x1, y1], [x2, y2], [x3, y3], etc]` – Each coordinate pair in the list represents a point on the boundary of the robot with the robot's center assumed to be at [0, 0]. The measurement units are meters. The points can be listed in either clockwise or counterclockwise order around the perimeter of the robot.
- `inflation_radius: 0.3` – The radius in meters that obstacles will be inflated in the map. If your robot is having trouble passing through narrow

  doorways or other tight spaces, trying reducing this value slightly. Conversely, if the robot keeps running into things, try increasing the value.

8.1.2.3 `global_costmap_params.yaml`

There are a few parameters in this file that you might experiment with depending on the power of your robot's CPU and the reliability of the network connection between the robot and your workstation:

- `global_frame: /map` – For the global cost map, we use the the map frame as the global frame.
- `robot_base_fame: /base_footprint` – This will usually be either `/base_link` or `/base_footprint`. For a TurtleBot it is `/base_footprint`.
- `update_frequency: 1.0` – The frequency in Hz of how often the global map is updated with sensor information. The higher this number, the greater the load on your computer's CPU. Especially for the global map, one can generally get away with a relatively small value of between 1.0 and 5.0.
- `publish_frequency: 0` – For a static global map, there is generally no need to continually publish it.
- `static_map: true` – This parameter and the next are always set to opposite values. The global map is usually static so we set `static_map` to `true`.
- `rolling_window: false` – The global map is generally not updated as the robot moves to we set this parameter to `false`.
- `transform_tolerance: 1.0` – Specifies the delay in seconds that is tolerated between the frame transforms in the `tf` tree. For typical wireless connections between the robot and the workstation, something on the order of 1.0 seconds works OK.

8.1.2.4 `local_costmap_params.yaml`

There are a few more parameters to be considered for the local cost map:

- `global_frame: /odom` – For the local cost map, we use the odometry frame as the global frame
- `robot_base_fame: /base_footprint` – This will usually be either `/base_link` or `/base_footprint`. For a TurtleBot it is `/base_footprint`.

- `update_frequency: 3.0` – How frequently (times per second) do we update the local map based on sensor data. For a really slow computer, you may have to reduce this value.
- `publish_frequency: 1.0` – We definitely want updates to the local map published so we set a non-zero value here. Once per second should be good enough unless your robot needs to move more quickly.
- `static_map: false` – This parameter and the next are always set to opposite values. The local map is dynamically updated with sensor data so we set `static_map` to `false`.
- `rolling_window: true` – The local map is updated within a rolling window defined by the next few parameters.
- `width: 6.0` – The x dimension in meters of the rolling map.
- `height: 6.0` – The y dimension in meters of the rolling map.
- `resolution: 0.01` – The resolution in meters of the rolling map. This should match the resolution set in the YAML file for your map (explained in a later section).
- `transform_tolerance: 1.0` – Specifies the delay in seconds that is tolerated between the frame transforms in the `tf` tree or the mapping process will temporarily abort. On a fast computer connected directly to the robot, a value 1.0 should work OK. But on slower computers and especially over wireless connections, the tolerance may have to be increased. The tell-tale warning message you will see when the tolerance is set too low will look like the following: