Robotics & Data Mining Summer School

Lesson 08. Robotics Operating System

Kirill Svyatov, Alexander Miheev

Ulyanovsk State Technical University,

Faculty of Information Systems and Technologies

What is tf?

- A robotic system typically has many coordinate frames that change over **time**, such as a world frame, base frame, gripper frame, head frame, etc.
- tf is a transformation system that allows making computations in one frame and then transforming them to another at any desired point in time
- tf allows you to ask questions like:
  - What is the current pose of the base frame of the robot in the map frame?
  - What is the pose of the object in my gripper relative to my base?
  - Where was the head frame relative to the world frame, 5 seconds ago?

Pros of tf

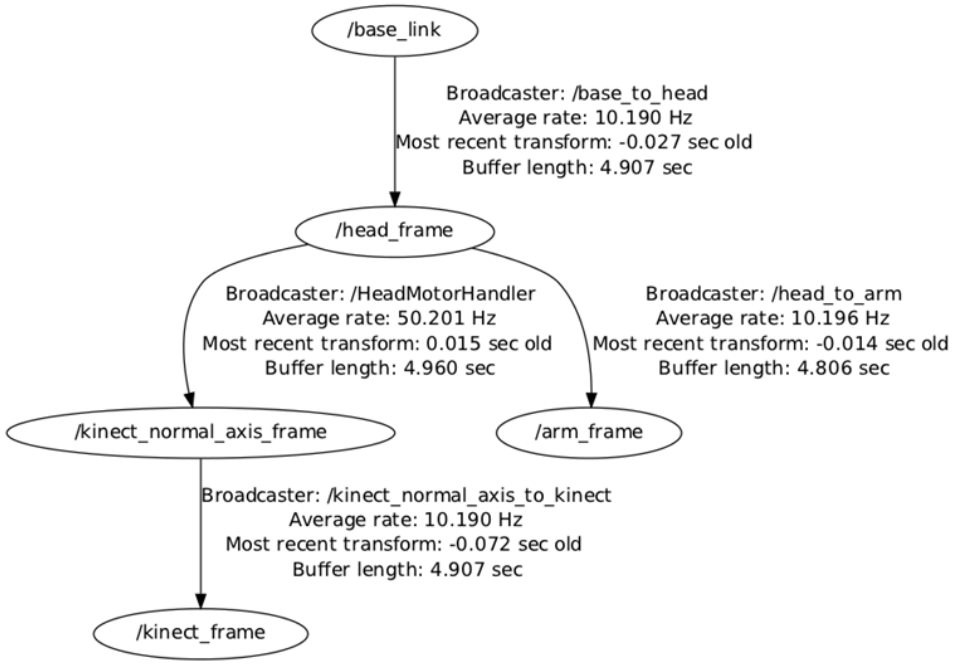
- Distributed system – no single point of failure
- No data loss when transforming multiple times
- No computational cost of intermediate data transformations between coordinate frames
- The user does not need to worry about which frame their data started
- Information about past locations is also stored and accessible (after local recording was started)

tf Nodes

- There are two types of tf nodes:
  - **Publishers** – publish transforms between coordinate frames on /tf
  - **Listeners** – listen to /tf and cache all data heard up to cache limit

Transform Tree

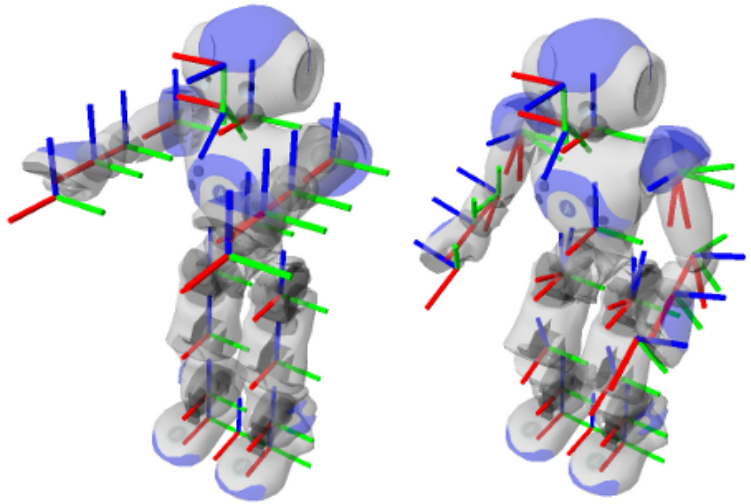
- TF builds a tree c

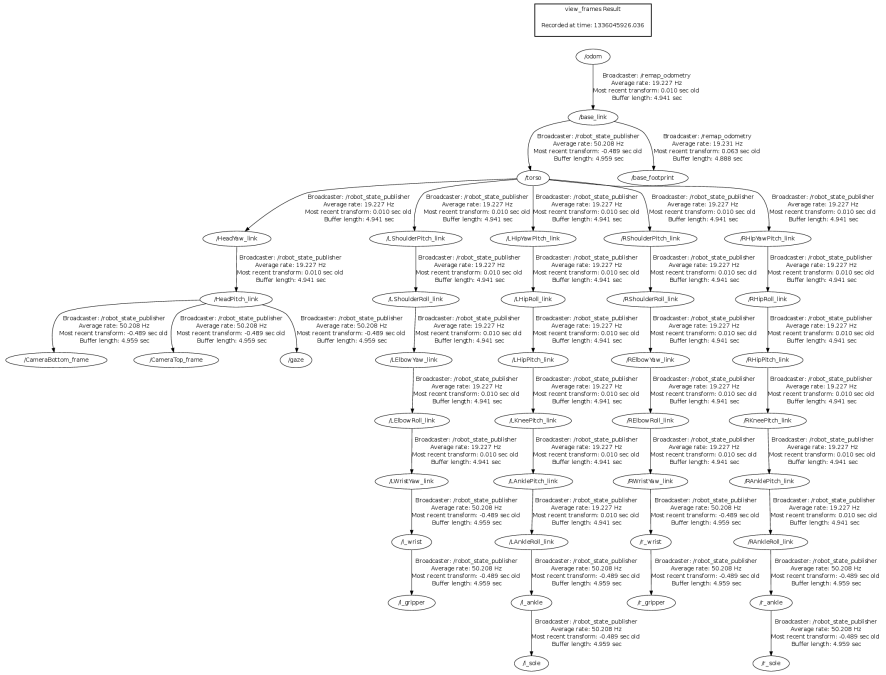


- Can support multiple disconnected trees
- Transforms only work within the same tree

Transform Tree Example

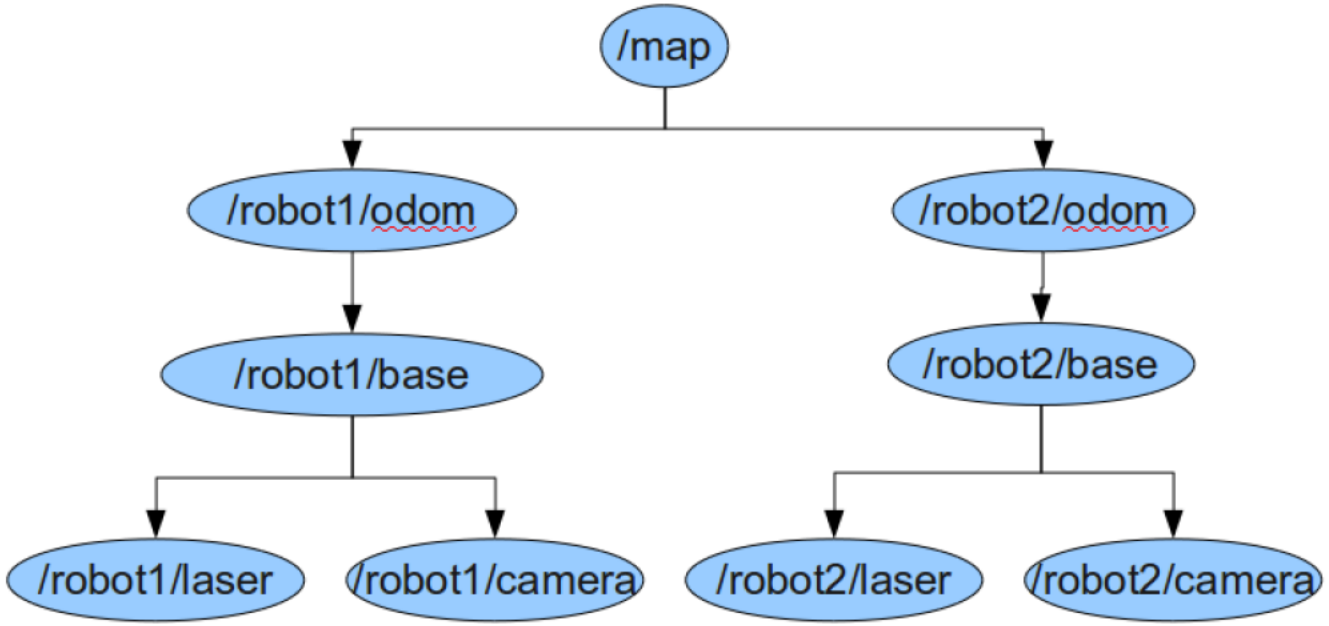
- Nao's TF



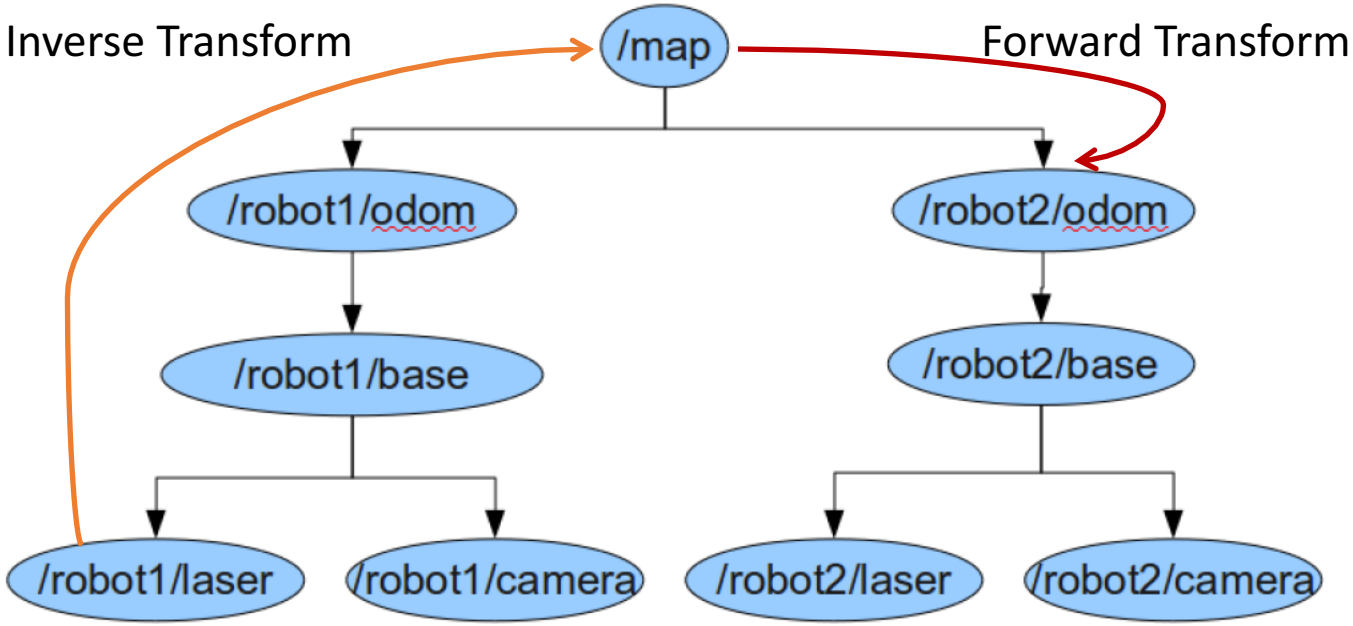


How To Use the TF Tree

- Given the following TF tree, let's say we want robot2 to navigate based on the laser data of robot1



How To Use the TF Tree



tf Demo

- Launch the turtle_tf_demo by typing:

```
$ roslaunch turtle_tf turtle_tf_demo.launch
```

- In another terminal run the turtle_tf_listener

```
$ rosrun turtle_tf turtle_tf_listener
```

- Now you should see a window with two turtles where one follows the other
- You can drive the center turtle around in the turtlesim using the keyboard arrow keys

```
$ rosrun turtlesim turtle_teleop_key
```

tf Demo

tf Demo

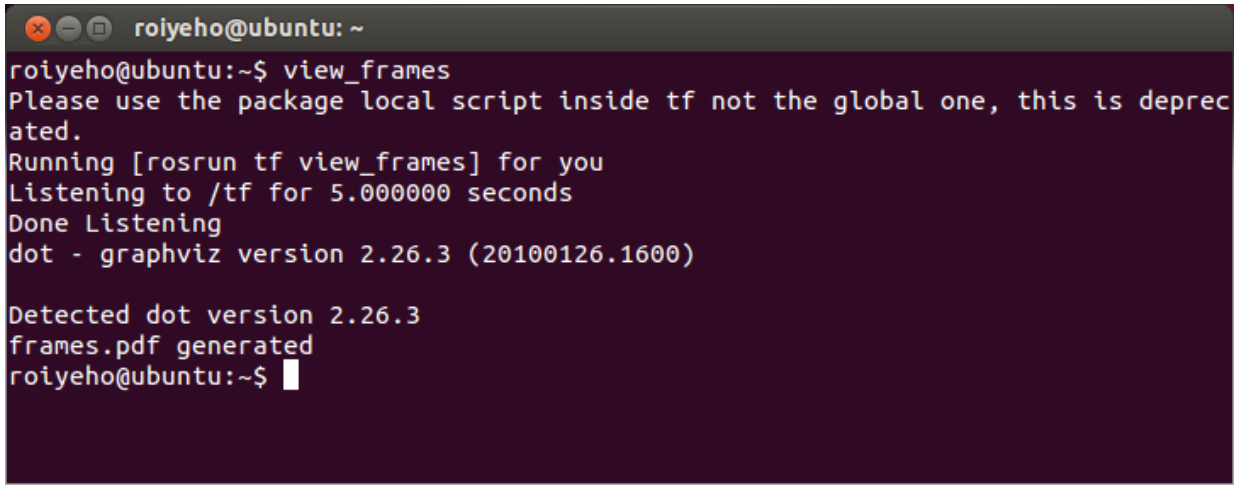
- This demo is using the tf library to create three coordinate frames: a world frame, a turtle1 frame, and a turtle2 frame.
- A **tf broadcaster** publishes the turtle coordinate frames and a **tf listener** computes the distance between the turtle frames to follow the other turtle

tf Command-line Tools

- view_frames: visualizes the full tree of coordinate transforms
- tf_monitor: monitors transforms between frames
- tf_echo: prints specified transform to screen
- roswtf: with the tfwtf plugin, helps you track down problems with tf
- static_transform_publisher is a command line tool for sending static transforms

view_frames

- view_frames creates a diagram of the frames being broadcast by tf over ROS

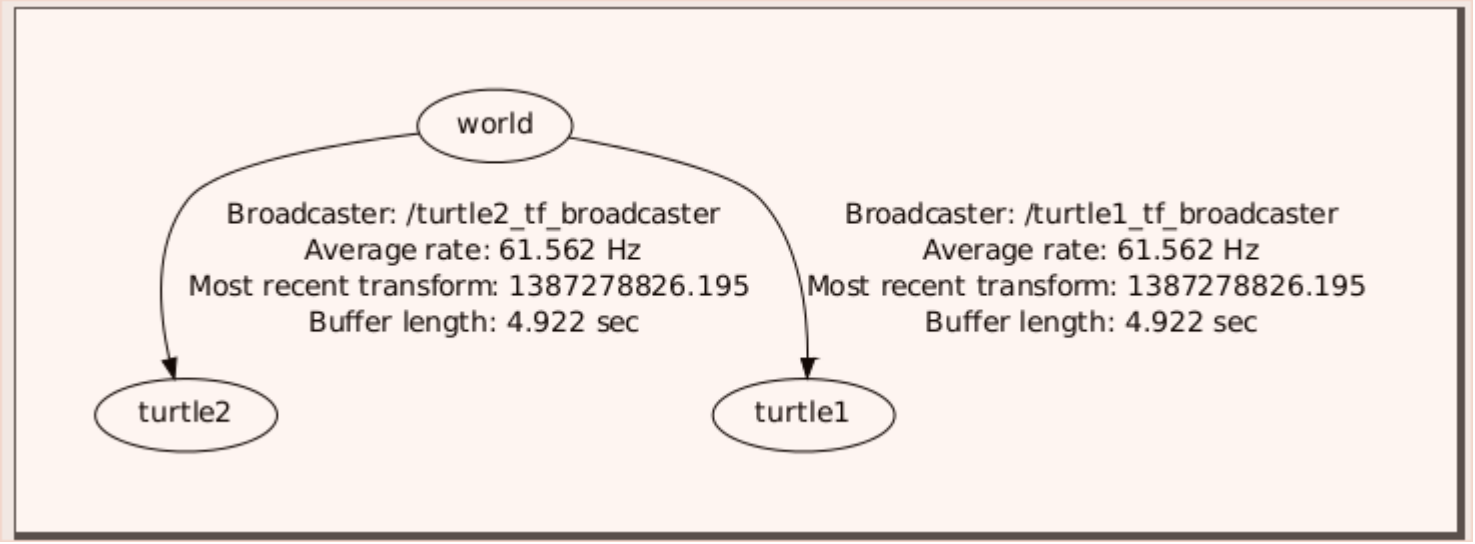


- To view the tree write:

```
$ evince frames.pdf
```

view_frames

- The TF tree:



tf_echo

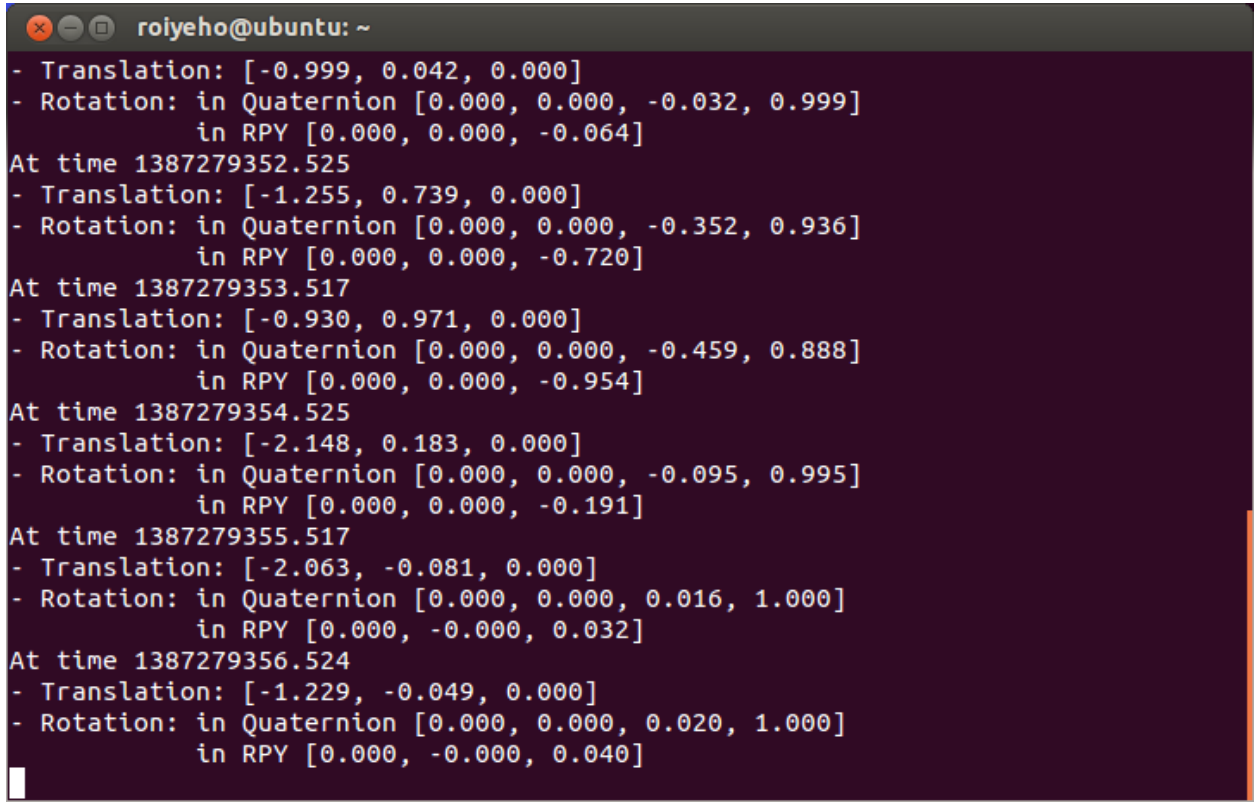
- tf_echo reports the transform between any two frames broadcast over ROS
- Usage:

```
$ rosrun tf tf_echo [reference_frame] [target_frame]
```

- Let's look at the transform of the turtle2 frame with respect to turtle1 frame which is equivalent to: $T_{turtle1\_turtle2} = T_{turtle1\_world} * T_{world\_turtle2}$

```
$ rosrun tf tf_echo turtle1 turtle2
```

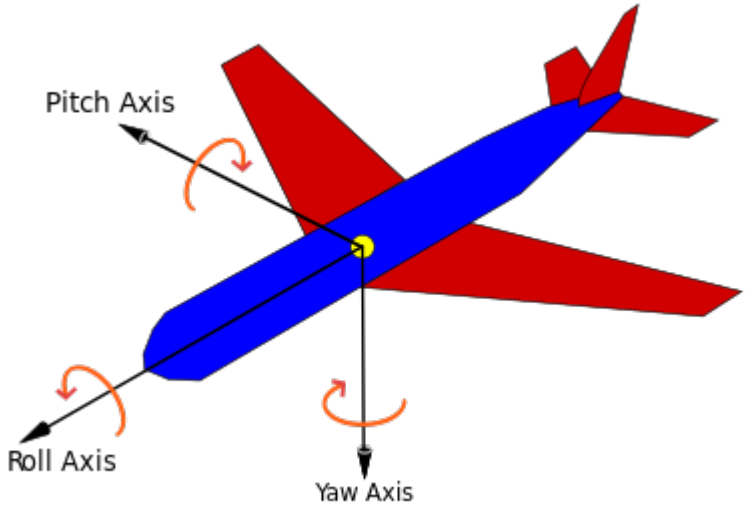
tf_echo

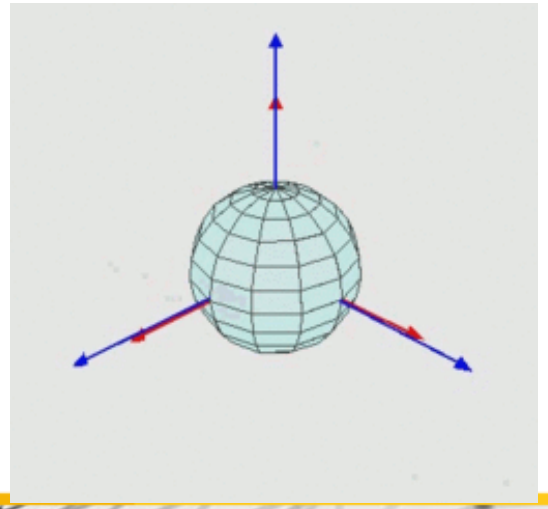


- As you drive your turtle around you will see the transform change as the two turtles move relative to each other

Rotation Representation

- There are many ways to represent rotations:
  - Euler angles yaw, pitch, and roll about Z, Y, X axes respectively
  - Rotation matrix
  - Quaternions





Quaternions

- In mathematics, quaternions are a number system that extends the complex numbers
- The fundamental formula for quaternion multiplication (Hamilton, 1843):

$$i^2 = j^2 = k^2 = ijk = -1$$

- Quaternions find uses in both theoretical and applied mathematics, in particular for calculations involving 3D rotations such as in computers graphics and computer vision

Quaternions and Spatial Rotation

- Any rotation in 3D can be represented as a combination of a vector u (the Euler axis) and a scalar θ (the rotation angle)
- A rotation with an angle of rotation θ around the axis defined by the unit vector

is represented by $\vec{u} = (u_x, u_y, u_z) = u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k}$

$$\mathbf{q} = e^{\frac{1}{2}\theta(u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})} = \cos\tfrac{1}{2}\theta + (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})\sin\tfrac{1}{2}\theta$$

Quaternions and Spatial Rotation

- Quaternions give a simple way to encode this axis–angle representation in 4 numbers
- Can apply the corresponding rotation to a position vector using a simple formula
  - http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation
- Advantages of using quaternions:
  - Nonsingular representation
    - there are 24 different possibilities to specify Euler angles
  - More compact (and faster) than matrices.

tf_monitor

- Print information about the current coordinate transform tree to console

```
$ rosrun tf tf_monitor
```

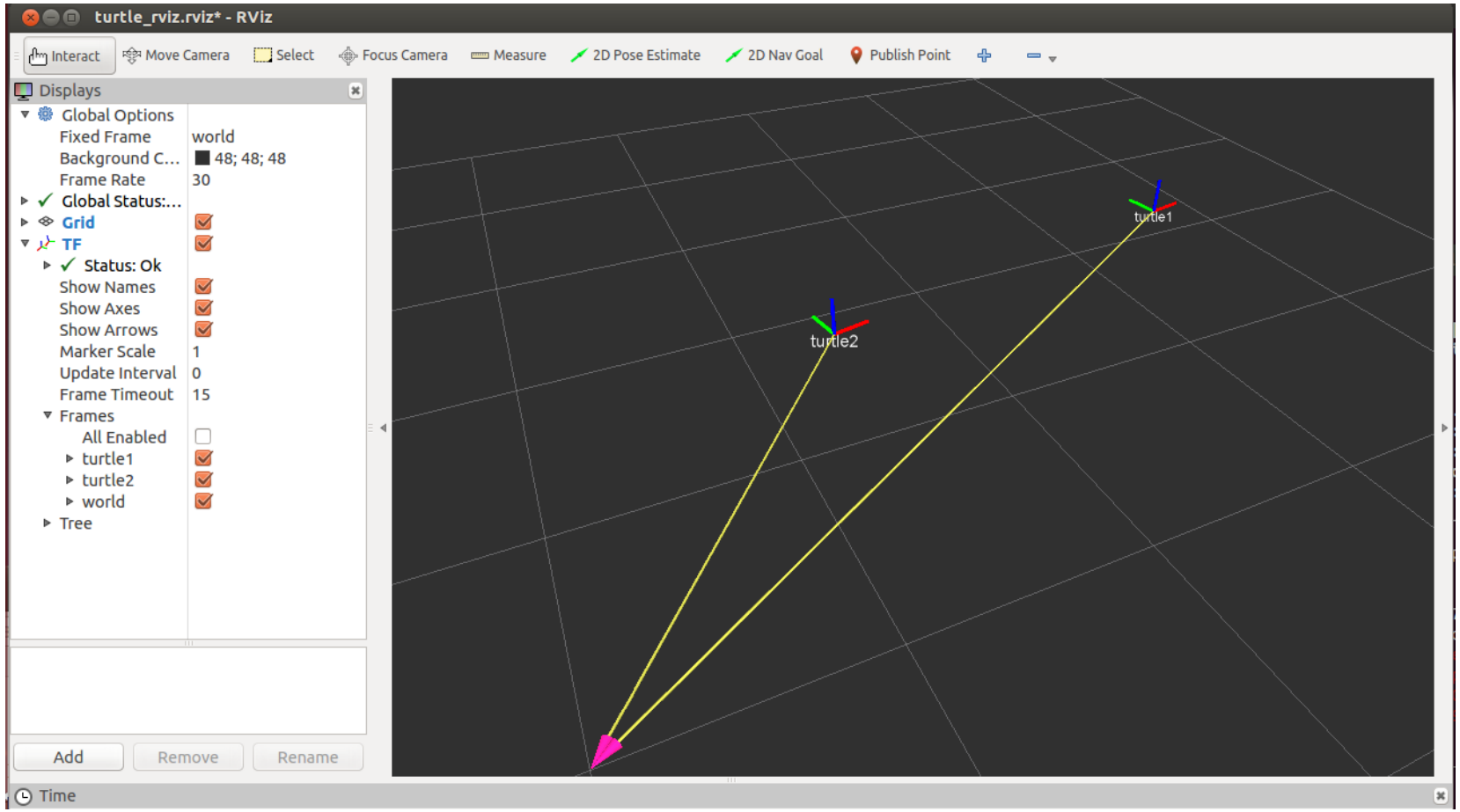
rviz and tf

- Let's look at our turtle frames using rviz
- Start rviz with the turtle_tf configuration file using the -d option for rviz:

```
$ rosrun rviz rviz -d `rospack find turtle_tf`/rviz/turtle_rviz.rviz
```

- On the left side bar you will see the frames broadcast by tf
- Note that the fixed frame is /world
  - The fixed frame is assumed not to be moving over time
- As you drive the turtle around you will see the frames change in rviz

rviz and tf


turtle_rviz.rviz* - RViz
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Fixed Frame
world
Background C...
48; 48; 48
Frame Rate
30
Global Status:...
Grid
TF
Status: Ok
Show Names
Show Axes
Show Arrows
Marker Scale
1
Update Interval
0
Frame Timeout
15
Frames
All Enabled
turtle1
turtle2
world
Tree
Add
Remove
Rename
Time
turtle1
turtle2

Broadcasting Transforms

- A tf broadcaster sends out the relative pose of coordinate frames to the rest of the system
- A system can have many broadcasters, each provides information about a different part of the robot
- We will now write the code to reproduce the tf demo

Writing a tf broadcaster

- First create a new package called tf_demo that depends on tf, roscpp, rospy and turtlesim

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg tf_demo tf roscpp rospy turtlesim
```

- Build the package by calling catkin_make
- Open the package in Eclipse and add a new source file called tf_broadcaster.cpp

tf_broadcaster.cpp (1)

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <turtlesim/Pose.h>

std::string turtle_name;

void poseCallback(const turtlesim::PoseConstPtr& msg)
{
    static tf::TransformBroadcaster br;
    tf::Transform transform;
    transform.setOrigin(tf::Vector3(msg->x, msg->y, 0.0));
    tf::Quaternion quaternion;
    transform.setRotation(tf::createQuaternionFromYaw(msg->theta));
    br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
"world", turtle_name));
}
```

tf_broadcaster.cpp (2)

```
int main(int argc, char** argv){
    ros::init(argc, argv, "my_tf_broadcaster");
    if (argc != 2) {
      ROS_ERROR("need turtle name as argument");
      return -1;
    };
    turtle_name = argv[1];

    ros::NodeHandle node;
    ros::Subscriber sub = node.subscribe(turtle_name + "/pose", 10,
&poseCallback);

    ros::spin();
    return 0;
};
```

Sending Transforms

```
br.sendTransform(tf::StampedTransform(transform,
ros::Time::now(), "world", turtle_name));
```

- Sending a transform with a TransformBroadcaster requires 4 arguments:
  - The transform object
  - A timestamp, usually we can just stamp it with the current time, ros::Time::now()
  - The name of the parent frame of the link we're creating, in this case "world"
  - The name of the child frame of the link we're creating, in this case this is the name of the turtle itself

Running the Broadcaster

- Create tf_demo.launch in the /launch subfolder

```
<launch>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <!-- tf broadcaster node -->
  <node pkg="tf_demo" type="turtle_tf_broadcaster"
      args="/turtle1" name="turtle1_tf_broadcaster" />
 </launch>
```

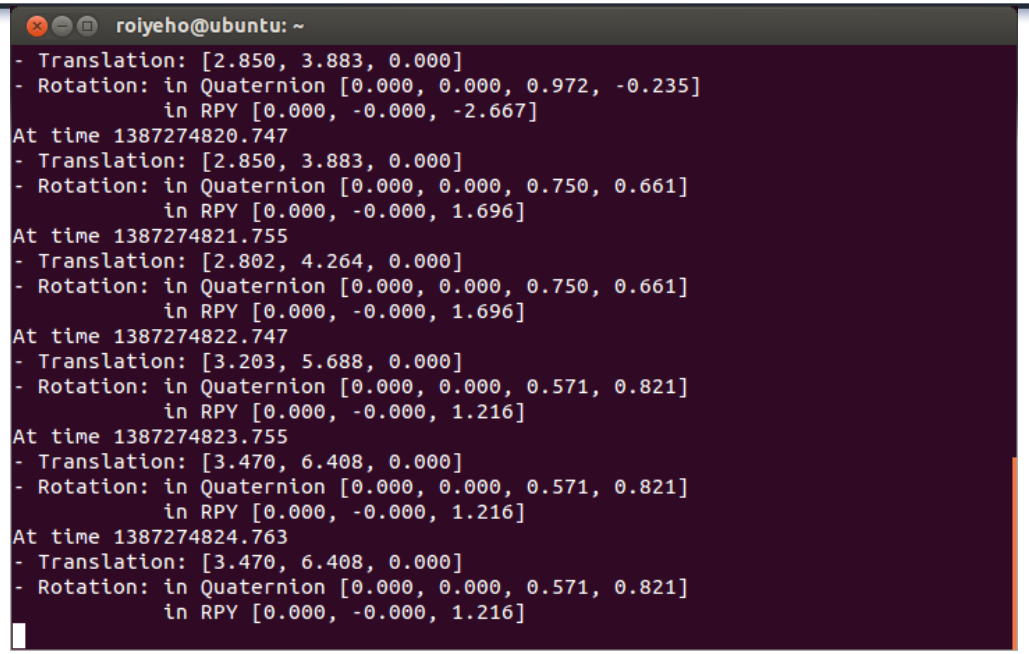
- Run the launch file

```
$ cd ~/catkin_ws/src
$ roslaunch tf_demo tf_demo.launch
```

Checking the Results

- Use the **tf_echo** tool to check if the turtle pose is actually getting broadcast to tf:

```
$ rosrun tf tf_echo /world /turtle1
```



Writing a tf listener

- A tf listener receives and buffers all coordinate frames that are broadcasted in the system, and queries for specific transforms between frames
- Next we'll create a tf listener that will listen to the transformations coming from the tf broadcaster
- Add tf_listener.cpp to your project with the following code

tf_listener.cpp (1)

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <turtlesim/Spawn.h>
#include <geometry_msgs/Twist.h>

int main(int argc, char** argv) {
    ros::init(argc, argv, "my_tf_listener");

    ros::NodeHandle node;

    ros::service::waitForService("spawn");
    ros::ServiceClient add_turtle =
       node.serviceClient<turtlesim::Spawn>("spawn");
    turtlesim::Spawn srv;
    add_turtle.call(srv);

    ros::Publisher turtle_vel =
       node.advertise<geometry_msgs::Twist>("turtle2/cmd_vel", 10);

    tf::TransformListener listener;
    ros::Rate rate(10.0);
```

tf_listener.cpp (2)

```
    while (node.ok()) {
        tf::StampedTransform transform;
        try {
            listener.waitForTransform("/turtle2", "/turtle1", ros::Time(0),
ros::Duration(10.0));
            listener.lookupTransform("/turtle2", "/turtle1", ros::Time(0),
transform);
        } catch (tf::TransformException ex) {
            ROS_ERROR("%s",ex.what());
        }

        geometry_msgs::Twist vel_msg;
        vel_msg.angular.z = 4 * atan2(transform.getOrigin().y(),
                                      transform.getOrigin().x());
        vel_msg.linear.x = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                      pow(transform.getOrigin().y(), 2));

        turtle_vel.publish(vel_msg);

        rate.sleep();
    }
    return 0;
};
```

Creating a TransformListener

- To use the TransformListener, we need to include the tf/transform_listener.h header file
- Once the listener is created, it starts receiving tf transformations over the wire, and buffers them for up to 10 seconds
- The TransformListener object should be scoped to persist otherwise its cache will be unable to fill and almost every query will fail
  - A common method is to make the TransformListener object a member variable of a class

Core Methods of TransformListener

- LookupTransform()
  - Get the transform between two coordinate frames
- WaitForTransform()
  - Block until timeout or transform is available
- CanTransform()
  - Test if a transform is possible between to coordinate frames

lookupTransform

```
listener.lookupTransform("/turtle2", "/turtle1",
ros::Time(0), transform);
```

- To query the listener for a specific transformation, you need to pass 4 arguments:
  - We want the transform from this frame ...
  - ... to this frame.
  - The time at which we want to transform. Providing ros::Time(0) will get us the latest available transform.
  - The object in which we store the resulting transform.

Running the Listener

- Add the following lines to CMakeLists.txt

```
add_executable(tf_listener src/tf_listener.cpp)
target_link_libraries(tf_listener
  ${catkin_LIBRARIES}
)
```

- Build the package by calling catkin_make

Launch File

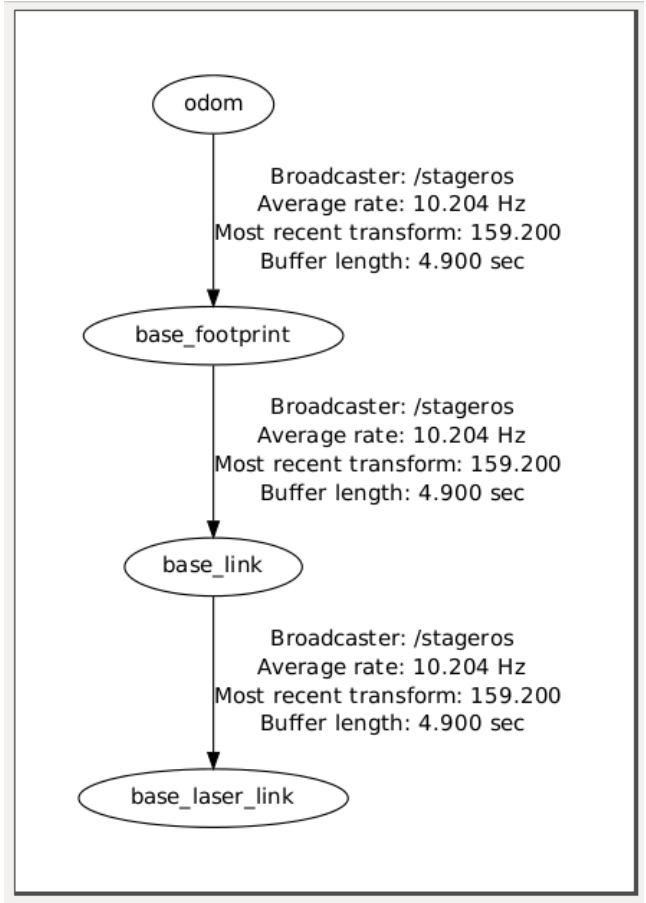
- Add the following lines to tf_demo.launch

```
<launch>
  <!-- Turtlesim Node-->
  <node pkg="turtlesim" type="turtlesim_node" name="sim"/>
  <node pkg="turtlesim" type="turtle_teleop_key" name="teleop" output="screen"/>

  <!-- tf broadcaster node -->
  <node pkg="tf_demo" type="tf_broadcaster"
      args="/turtle1" name="turtle1_tf_broadcaster" />

  <!-- Second broadcaster node -->
  <node pkg="tf_demo" type="tf_broadcaster"
      args="/turtle2" name="turtle2_tf_broadcaster" />

  <!-- tf listener node -->
  <node pkg="tf_demo" type="tf_listener" name="listener" />

 </launch>
```

Check the Results

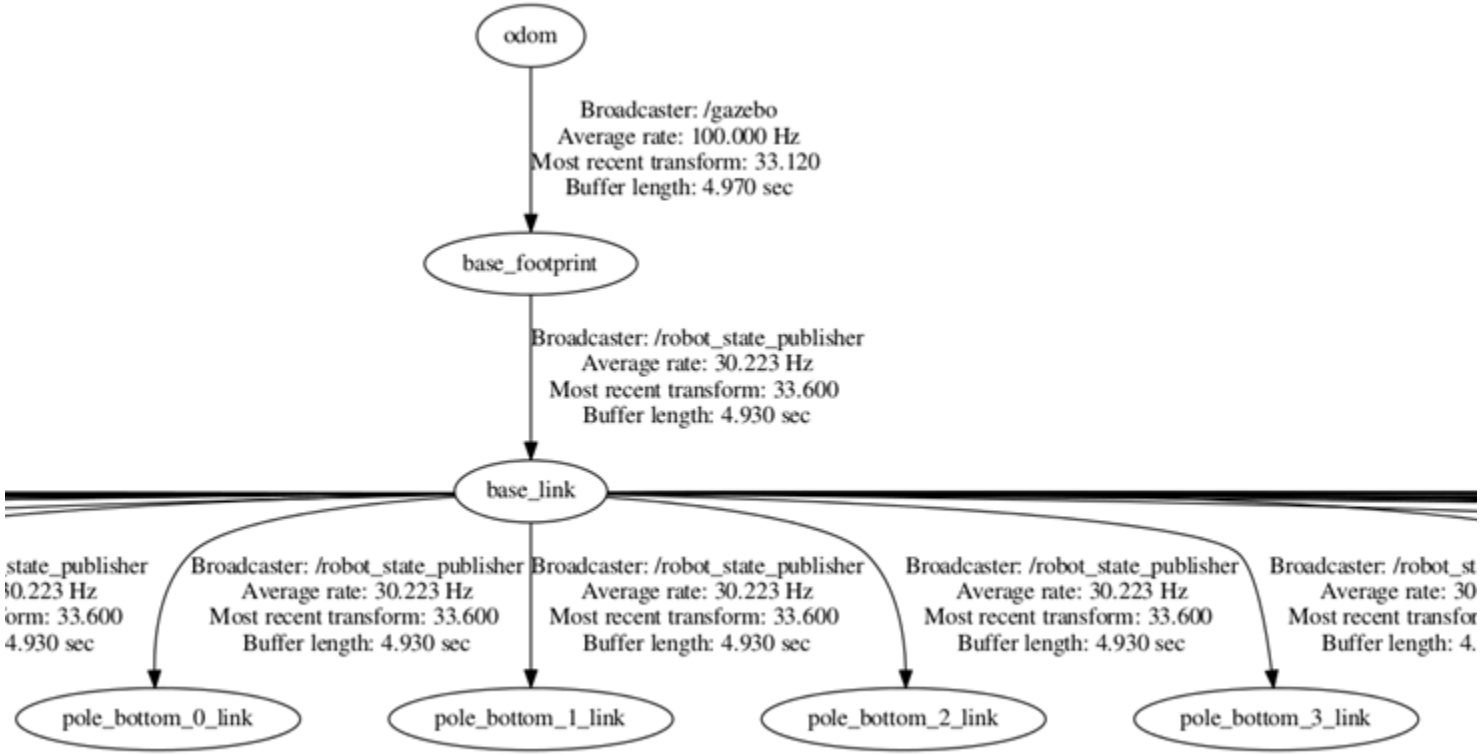
- To see if things work, simply drive around the first turtle using the arrow keys (make sure your terminal window is active, not your simulator window), and you'll see the second turtle following the first one!

Typical TF Frames



- **odom** – the self consistent coordinate frame using the odometry measurements only
- **base_footprint** – the base of the robot at zero height above the ground
- **base_link** – the base link of the robot, placed at the rotational center of the robot
- **base_laser_link** – the location of the laser sensor

Turtlebot TF Frames



Find Robot Location

- We'll now see an example how to use tf to determine the robot's current location in the world
- First, we would like to change the TurtleBot initial location in Gazebo (default is x=0,y=0)
- You can change the initial location in by setting the environment variable ROBOT_INITIAL_POSE, e.g.:

```
$ export ROBOT_INITIAL_POSE="-x -1 -y -2"
```

- To get robot's location in its own coordinate frame (i.e., relative to its starting location on the map) create a TF listener from the /base_footprint to the /odom frame

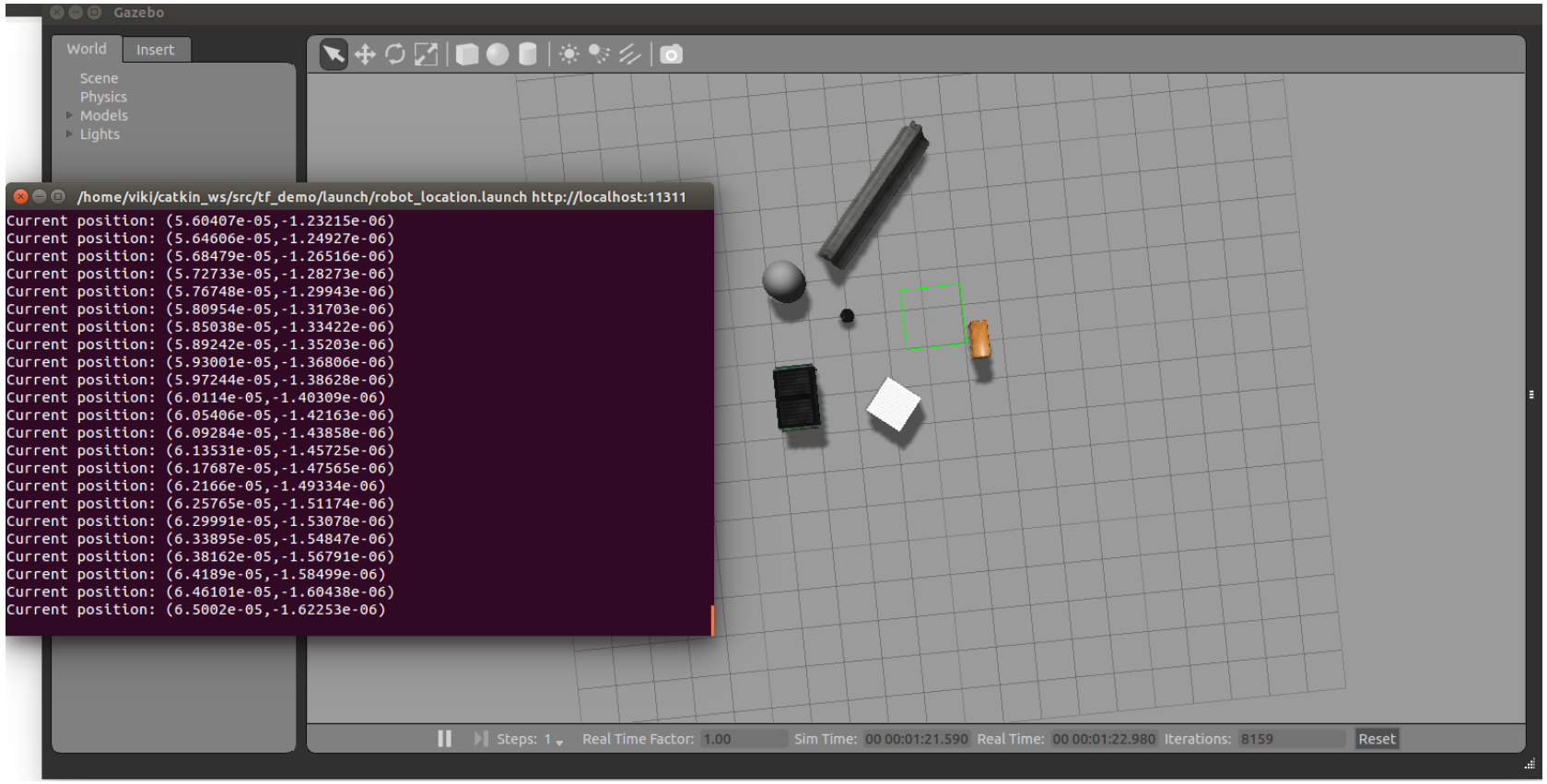
robot_location.cpp (1)

```
#include <ros/ros.h>
#include <tf/transform_listener.h>

using namespace std;

int main(int argc, char** argv){
    ros::init(argc, argv, "robot_location");
    ros::NodeHandle node;

    tf::TransformListener listener;
    ros::Rate rate(2.0);
    listener.waitForTransform("/odom", "/base_footprint", ros::Time(0),
ros::Duration(10.0));
```

robot_location.cpp (2)

```
    while (ros::ok()){
        tf::StampedTransform transform;
        try {
            listener.lookupTransform("/odom", "/base_footprint", ros::Time(0),
transform);
            double x = transform.getOrigin().x();
            double y = transform.getOrigin().y();
            cout << "Current position: (" << x << "," << y << ")" << endl;
        } catch (tf::TransformException &ex) {
            ROS_ERROR("%s",ex.what());
        }
        rate.sleep();
    }

    return 0;
}
```

Find Robot Location



Static Transform Publisher

- In order to get the robot's location in the global coordinate frame, an map->odom transform needs to be published by some node
- This transformation is typically published by one of ROS mapping or localization nodes (next lesson)
- When assuming perfect localization of the robot, you can publish a static (fixed) transform between these frames

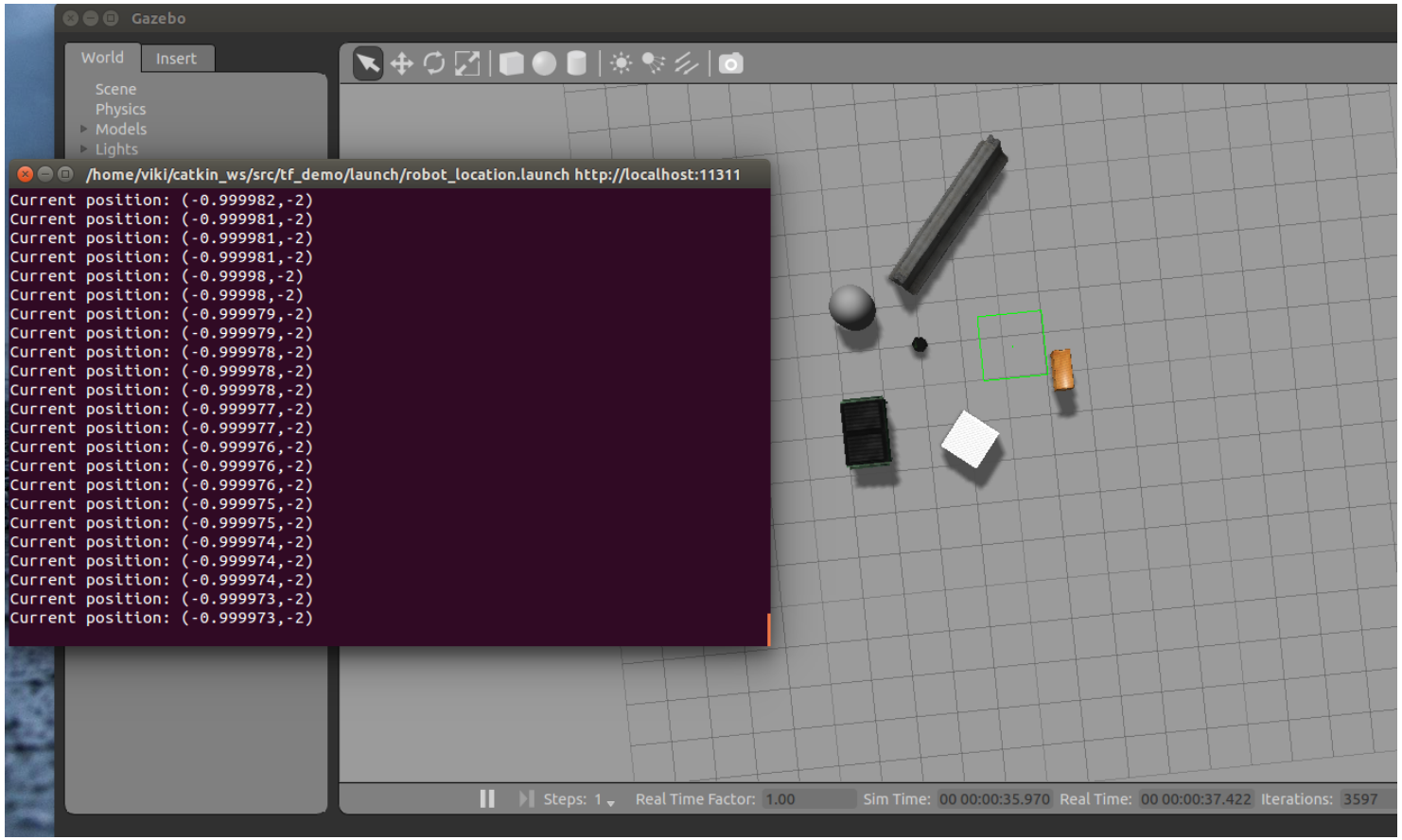
```
<launch>
  <!-- Publish a static transformation between /map and /odom -->
  <node name="tf" pkg="tf" type="static_transform_publisher" args="-1 -2 0
0 0 0 /map /odom 100" />
</launch>
```

Find Robot Location

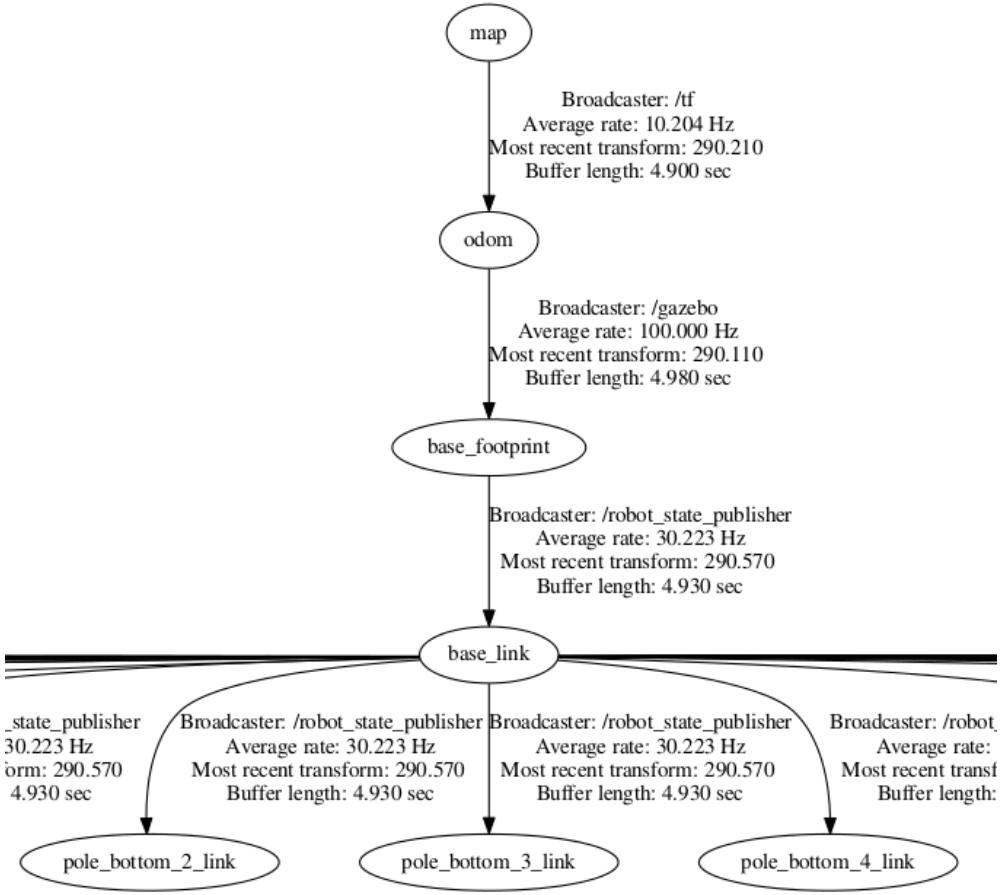
- Change the TF listener to listen to the transform from /base_footprint /map in order to get the robot's location in the map's frame

```
    while (ros::ok()){
        tf::StampedTransform transform;
        try {
            listener.lookupTransform("/map", "/base_footprint", ros::Time(0),
transform);
            double x = transform.getOrigin().x();
            double y = transform.getOrigin().y();
            cout << "Current position: (" << x << "," << y << ")" << endl;
        } catch (tf::TransformException &ex) {
            ROS_ERROR("%s",ex.what());
        }
        rate.sleep();
    }
    return 0;
}
```

Find Robot Location



New TF Tree


map
Broadcaster: /tf
Average rate: 10.204 Hz
Most recent transform: 290.210
Buffer length: 4.900 sec
odom
Broadcaster: /gazebo
Average rate: 100.000 Hz
Most recent transform: 290.110
Buffer length: 4.980 sec
base_footprint
Broadcaster: /robot_state_publisher
Average rate: 30.223 Hz
Most recent transform: 290.570
Buffer length: 4.930 sec
base_link
_state_publisher
30.223 Hz
orm: 290.570
4.930 sec
Broadcaster: /robot_state_publisher
Average rate: 30.223 Hz
Most recent transform: 290.570
Buffer length: 4.930 sec
Broadcaster: /robot_state_publisher
Average rate: 30.223 Hz
Most recent transform: 290.570
Buffer length: 4.930 sec
Broadcaster: /robot
Average rate:
Most recent transf
Buffer length:
pole_bottom_2_link
pole_bottom_3_link
pole_bottom_4_link

Final Launch File

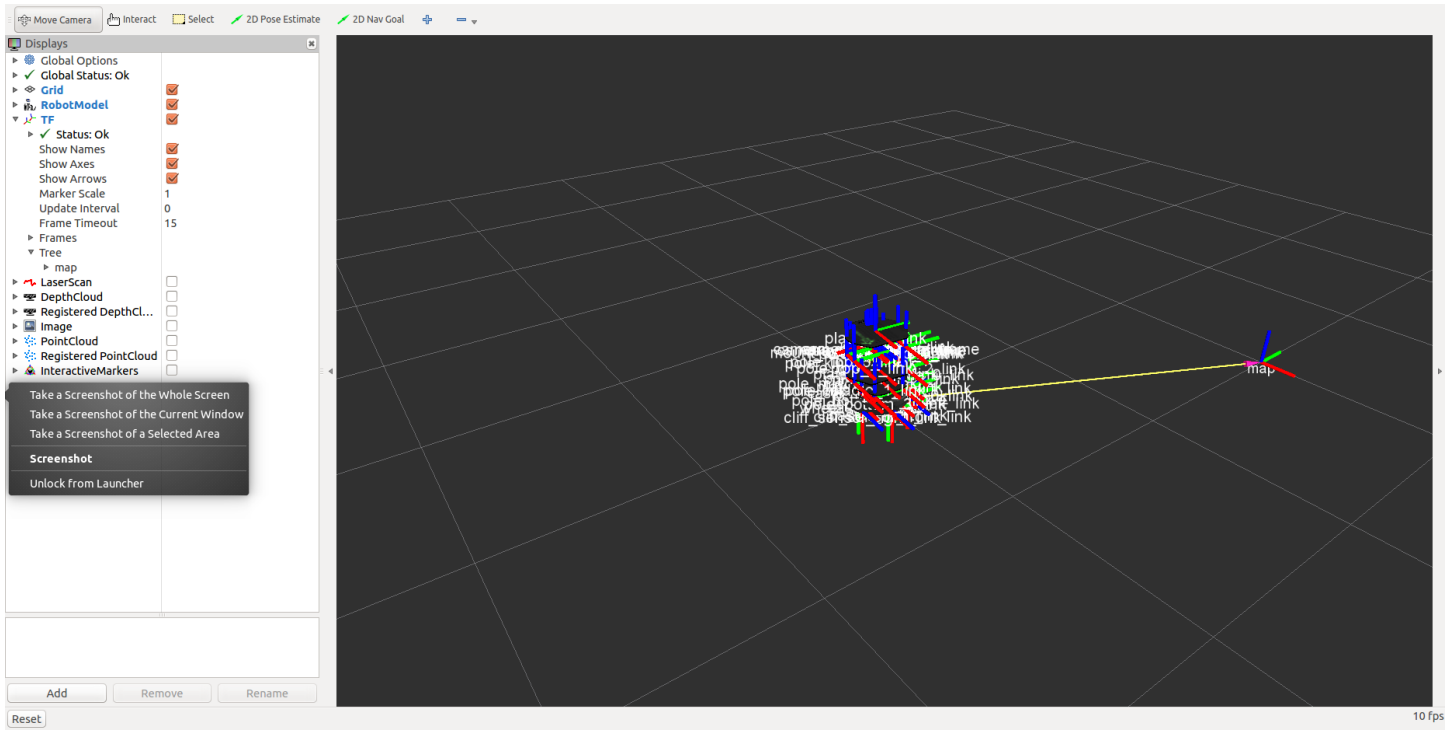
```
<launch>
  <param name="/use_sim_time" value="true"/>

  <!-- Run Gazebo with turtlebot -->
  <include file="$(find turtlebot_gazebo)/launch/turtlebot_world.launch"/>

  <!-- Publish a static transformation between /odom and /map -->
  <node name="tf" pkg="tf" type="static_transform_publisher" args="-1 -2 0 0 0 0 /map /odom
100" />

  <!– Run node -->
  <node name="robot_location" pkg="tf_demo" type="robot_location" output="screen" />
</launch>
```

Watch the TF Frames in rviz

- Run rviz

```
$ roslaunch turtlebot_rviz_launchers view_robot.launch
```

- Click the TF display checkbox



Ex. 6

- Write functions that translate the robot's current position in the world (x, y) to the cell in the grid map
  (i, j) in which the robot is located and vice versa
  - that take into account the map resolution
- Print the initial cell that the robot is located at the start

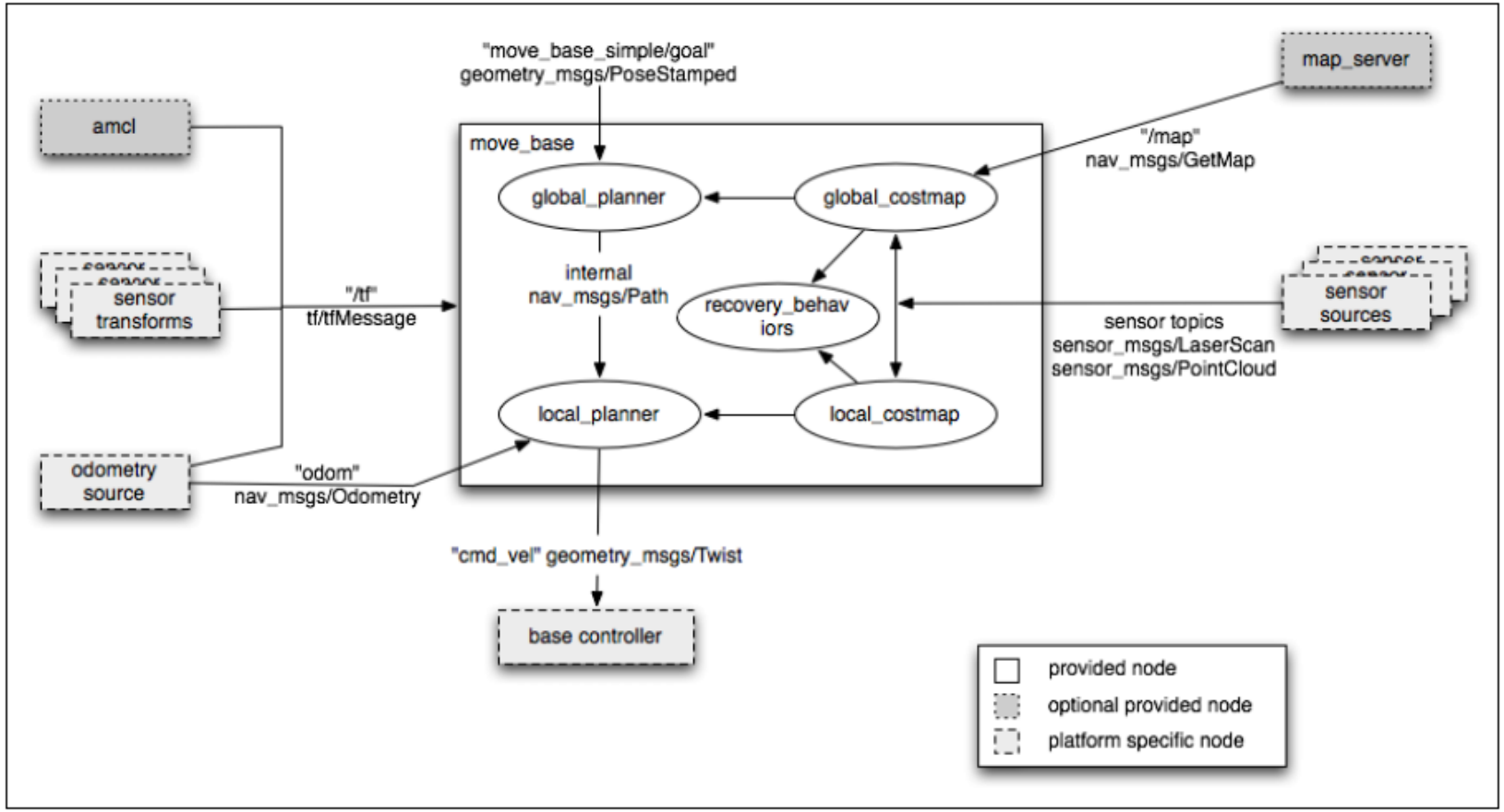
Robot Navigation

- One of the most basic things that a robot can do is to move around the world.
- To do this effectively, the robot needs to know where it is and where it should be going
- This is usually achieved by giving the robot a map of the world, a starting location, and a goal location
- In the previous lesson, we saw how to build a map of the world from sensor data.
- Now, we'll look at how to make your robot autonomously navigate from one part of the world to another, using this map and the ROS navigation packages

ROS Navigation Stack

- http://wiki.ros.org/navigation
- The goal of the navigation stack is to move a robot from one position to another position safely (without crashing or getting lost)
- It takes in information from the odometry and sensors, and a goal pose and outputs safe velocity commands that are sent to the robot
- ROS Navigation Introductory Video

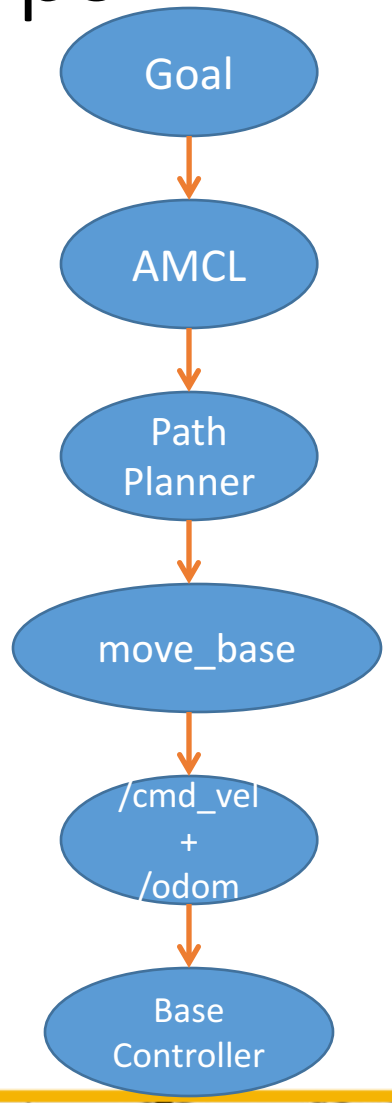
ROS Navigation Stack



Navigation Stack Main Components

Package/Component	Description
map_server	offers map data as a ROS Service
gmapping	provides laser-based SLAM
amcl	a probabilistic localization system
global_planner	implementation of a fast global planner for navigation
local_planner	implementations of the Trajectory Rollout and Dynamic Window approaches to local robot navigation
move_base	links together the global and local planner to accomplish the navigation task

Navigation Main Steps



Install Navigation Stack

- The navigation stack is not part of the standard ROS Indigo installation
- To install the navigation stack type:

```
$ sudo apt-get install ros-kinetic-navigation
```

Navigation Stack Requirements

Three main hardware requirements

- The navigation stack can only handle a differential drive and holonomic wheeled robots
  - It can also do certain things with biped robots, such as localization, as long as the robot does not move sideways
- A planar laser must be mounted on the mobile base of the robot to create the map and localization
  - Alternatively, you can generate something equivalent to laser scans from other sensors (Kinect for example)
- Its performance will be best on robots that are nearly square or circular

Navigation Planners

- Our robot will move through the map using two types of navigation—global and local
- The **global planner** is used to create paths for a goal in the map or a far-off distance
- The **local planner** is used to create paths in the nearby distances and avoid obstacles

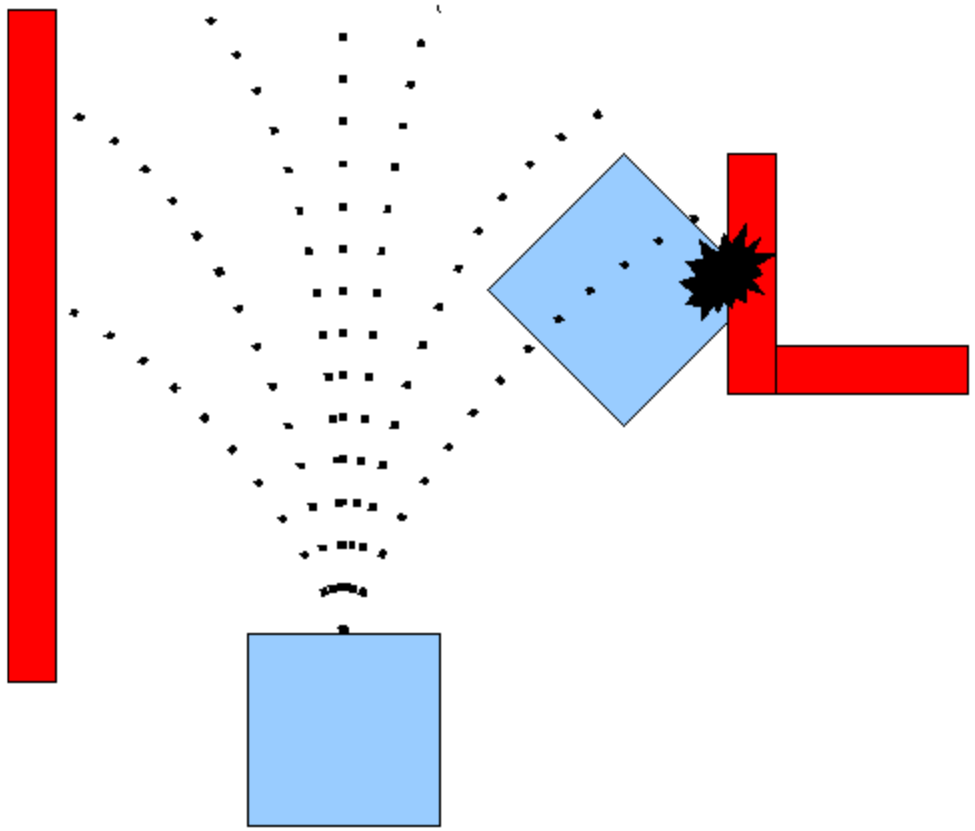
Global Planner

- NavFn provides a fast interpolated navigation function that creates plans for a mobile base
- The global plan is computed before the robot starts moving toward the next destination
- The planner operates on a costmap to find a minimum cost plan from a start point to an end point in a grid, using Dijkstra's algorithm
- The global planner generates a series of waypoints for the local planner to follow

Local Planner

- Chooses appropriate velocity commands for the robot to traverse the current segment of the global path
- Combines sensory and odometry data with both global and local cost maps
- Can recompute the robot's path on the fly to keep the robot from striking objects yet still allowing it to reach its destination
- Implements the **Trajectory Rollout and Dynamic Window** algorithm

Trajectory Rollout Algorithm



Taken from ROS Wiki http://wiki.ros.org/base_local_planner

Trajectory Rollout Algorithm

1. Discretely sample in the robot's control space (dx,dy,dθ)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time
3. Evaluate each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed
4. Discard illegal trajectories (those that collide with obstacles)
5. Pick the highest-scoring trajectory and send the associated velocity to the mobile base
6. Rinse and repeat

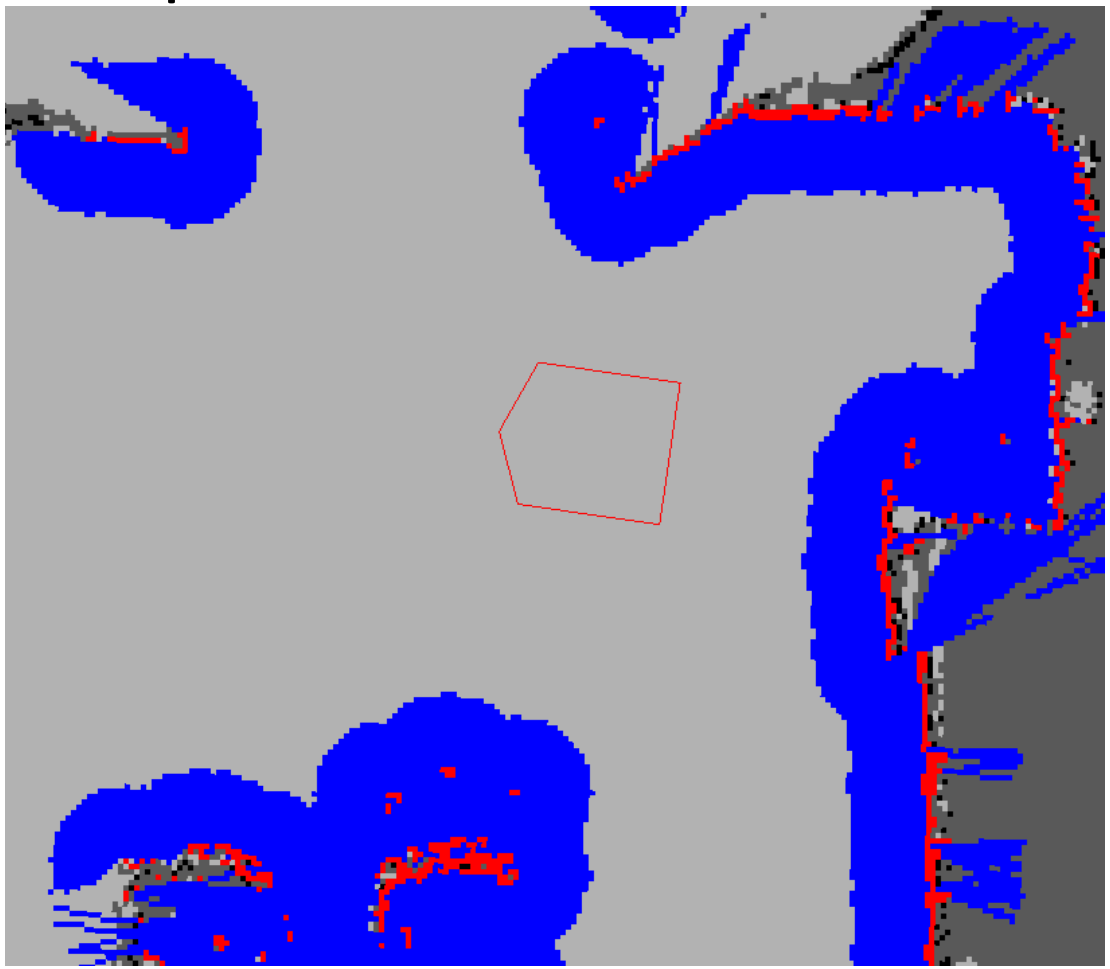
Local Planner Parameters

- The file **base_local_planner.yaml** contains a large number of ROS Parameters that can be set to customize the behavior of the base local planner
- Grouped into several categories:
  - robot configuration
  - goal tolerance
  - forward simulation
  - trajectory scoring
  - oscillation prevention
  - global plan

Costmap

- A data structure that represents places that are safe for the robot to be in a grid of cells
- It is based on the occupancy grid map of the environment and user specified inflation radius
- There are two types of costmaps in ROS:
  - **Global costmap** is used for global navigation
  - **Local costmap** is used for local navigation
- Each cell in the costmap has an integer value in the range [0 (FREE_SPACE), 255 (UNKNOWN)]
- Managed by the costmap_2d package

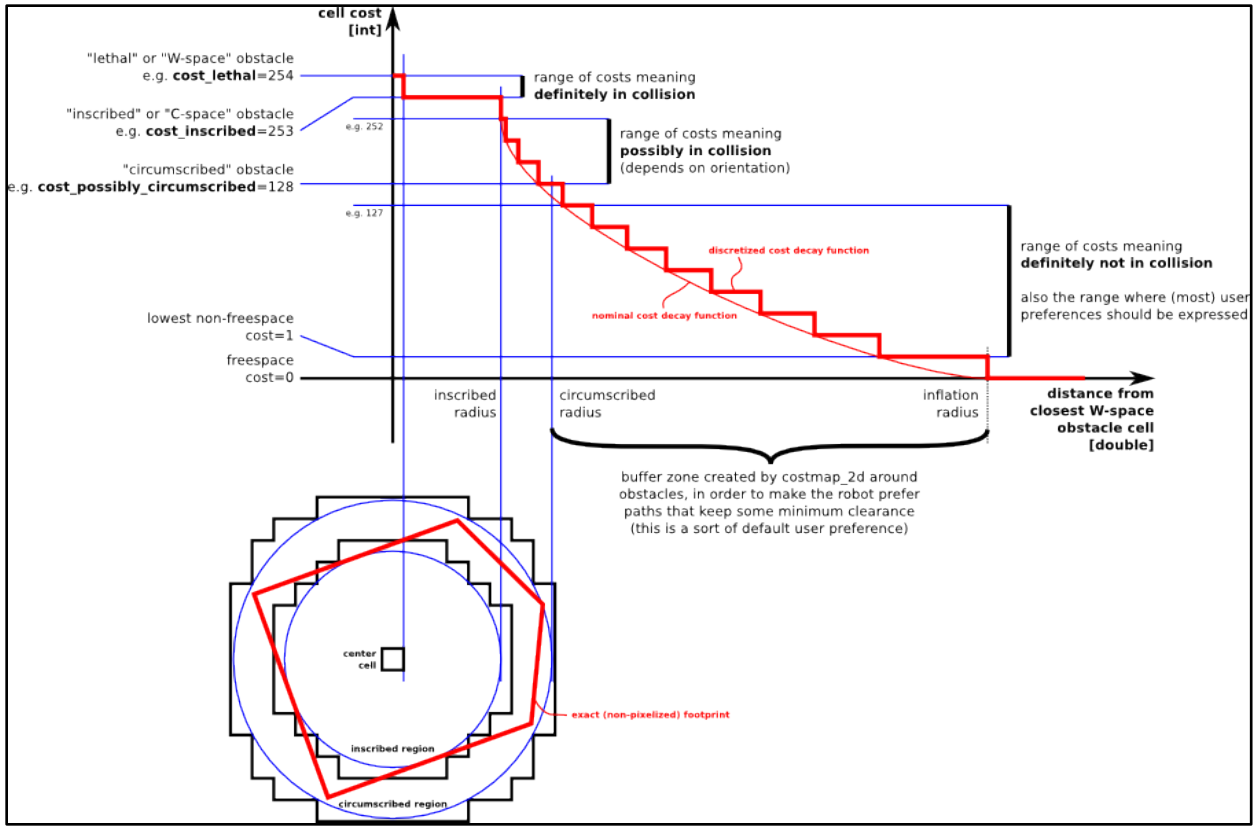
Costmap Example



Taken from ROS Wiki http://wiki.ros.org/costmap_2d

Inflation

- Inflation is the process of propagating cost values out from occupied cells that decrease with distance



Map Updates

- The costmap performs map update cycles at the rate specified by the **update_frequency** parameter
- In each cycle:
  - sensor data comes in
  - marking and clearing operations are perfomed in the underlying occupancy structure of the costmap
  - this structure is projected into the costmap where the appropriate cost values are assigned as described above
  - obstacle inflation is performed on each cell with a LETHAL_OBSTACLE value
    - This consists of propagating cost values outwards from each occupied cell out to a user-specified inflation radius

Costmap Parameters Files

- Configuration of the costmaps consists of three files:
  - costmap_common_params.yaml
  - global_costmap_params.yaml
  - local_costmap_params.yaml
- http://wiki.ros.org/costmap_2d/hydro/obstacles

Localization

- Localization is the problem of estimating the pose of the robot relative to a map
- Localization is not terribly sensitive to the exact placement of objects so it can handle small changes to the locations of objects
- ROS uses the **amcl** package for localization

AMCL

- amcl is a probabilistic localization system for a robot moving in 2D
- It implements the adaptive **Monte Carlo localization** approach, which uses a particle filter to track the pose of a robot against a known map
- The algorithm and its parameters are described in the book **Probabilistic Robotics** by Thrun, Burgard, and Fox (http://www.probabilistic-robotics.org/)
- Currently amcl works only with laser scans
  - However, it can be extended to work with other sensors

AMCL

- amcl takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates
- Subscribed topics:
  - scan – Laser scans
  - tf – Transforms
  - initialpose – Mean and covariance with which to (re-) initialize the particle filter
  - map – the map used for laser-based localization
- Published topics:
  - amcl_pose – Robot's estimated pose in the map, with covariance.
  - Particlecloud – The set of pose estimates being maintained by the filter

move_base

- The move_base package lets you move a robot to desired positions using the navigation stack
- The move_base node links together a global and local planner to accomplish its navigation task
- It may optionally perform recovery behaviors when the robot perceives itself as stuck

TurtleBot Navigation

- turtlebot_navigation package includes demos of map building using gmapping and localization with amcl, while running the navigation stack
- In param subdirectory it contains configuration files for TurtleBot navigation

Navigation Configuration Files

Configuration File	Description
global_planner_params.yaml	global planner configuration
navfn_global_planner_params.yaml	navfn configuration
dwa_local_planner_params.yaml	local planner configuration
costmap_common_params.yaml global_costmap_params.yaml local_costmap_params.yaml	costmap configuration files
move_base_params.yaml	move base configuration
amcl.launch.xml	amcl configuration

Autonomous Navigation of a Known Map

- amcl_demo.launch - launch file for navigation demo

```
<launch>
  <!-- Map server -->
  <arg name="map_file" default="$(env TURTLEBOT_GAZEBO_MAP_FILE)"/>
  <node name="map_server" pkg="map_server" type="map_server" args="$(arg map_file)" />

  <!-- Localization -->
  <arg name="initial_pose_x" default="0.0"/>
  <arg name="initial_pose_y" default="0.0"/>
  <arg name="initial_pose_a" default="0.0"/>
  <include file="$(find turtlebot_navigation)/launch/includes/amcl.launch.xml">
    <arg name="initial_pose_x" value="$(arg initial_pose_x)"/>
    <arg name="initial_pose_y" value="$(arg initial_pose_y)"/>
    <arg name="initial_pose_a" value="$(arg initial_pose_a)"/>
  </include>

  <!-- Move base -->
  <include file="$(find turtlebot_navigation)/launch/includes/move_base.launch.xml"/>
</launch>
```

Autonomous Navigation of a Known Map

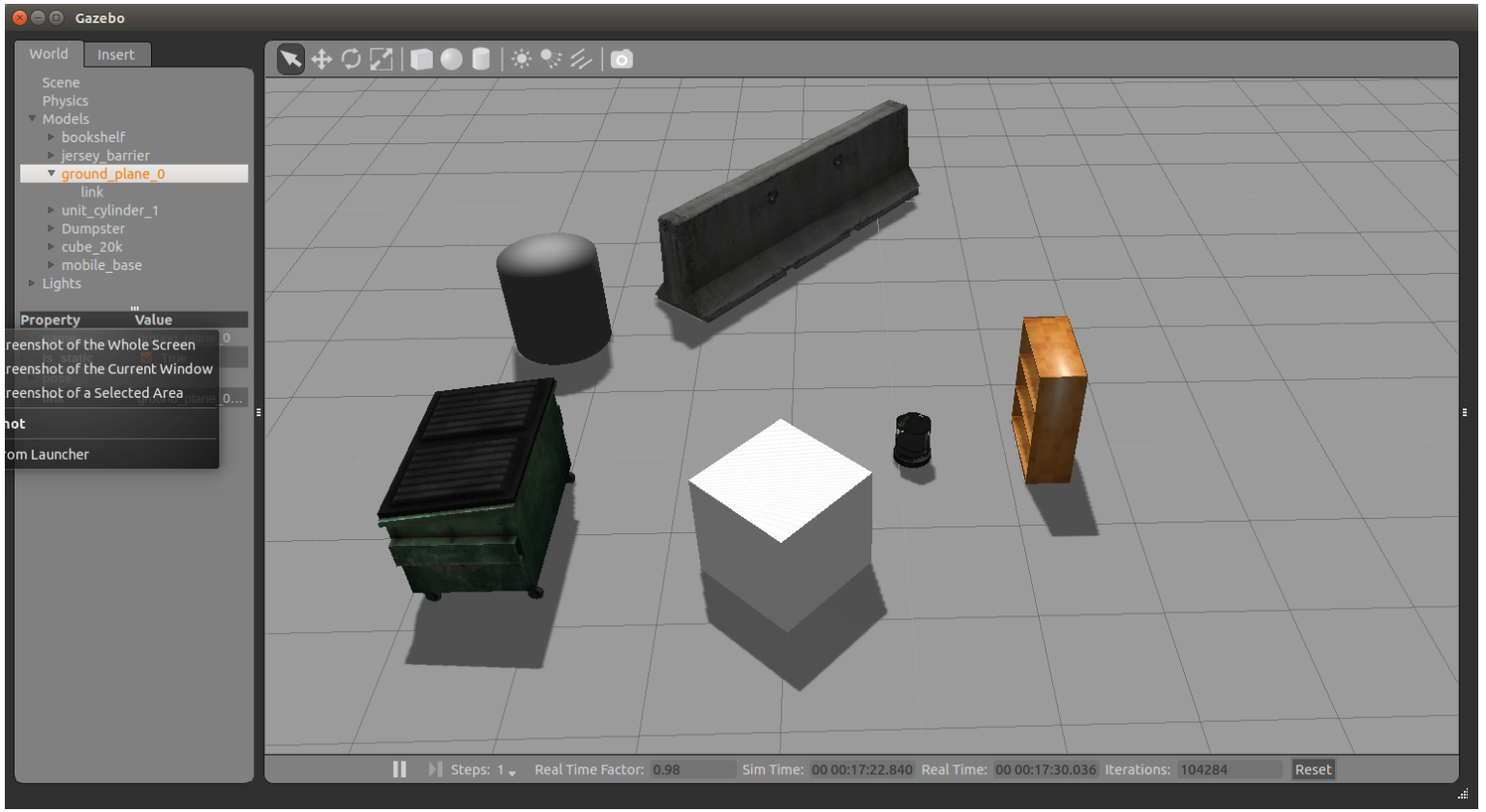
- Launch Gazebo with turtlebot

```
$ roslaunch turtlebot_gazebo turtlebot_world.launch
```

- Run the navigation demo

```
$ roslaunch turtlebot_gazebo amcl_demo.launch
```

Autonomous Navigation of a Known Map

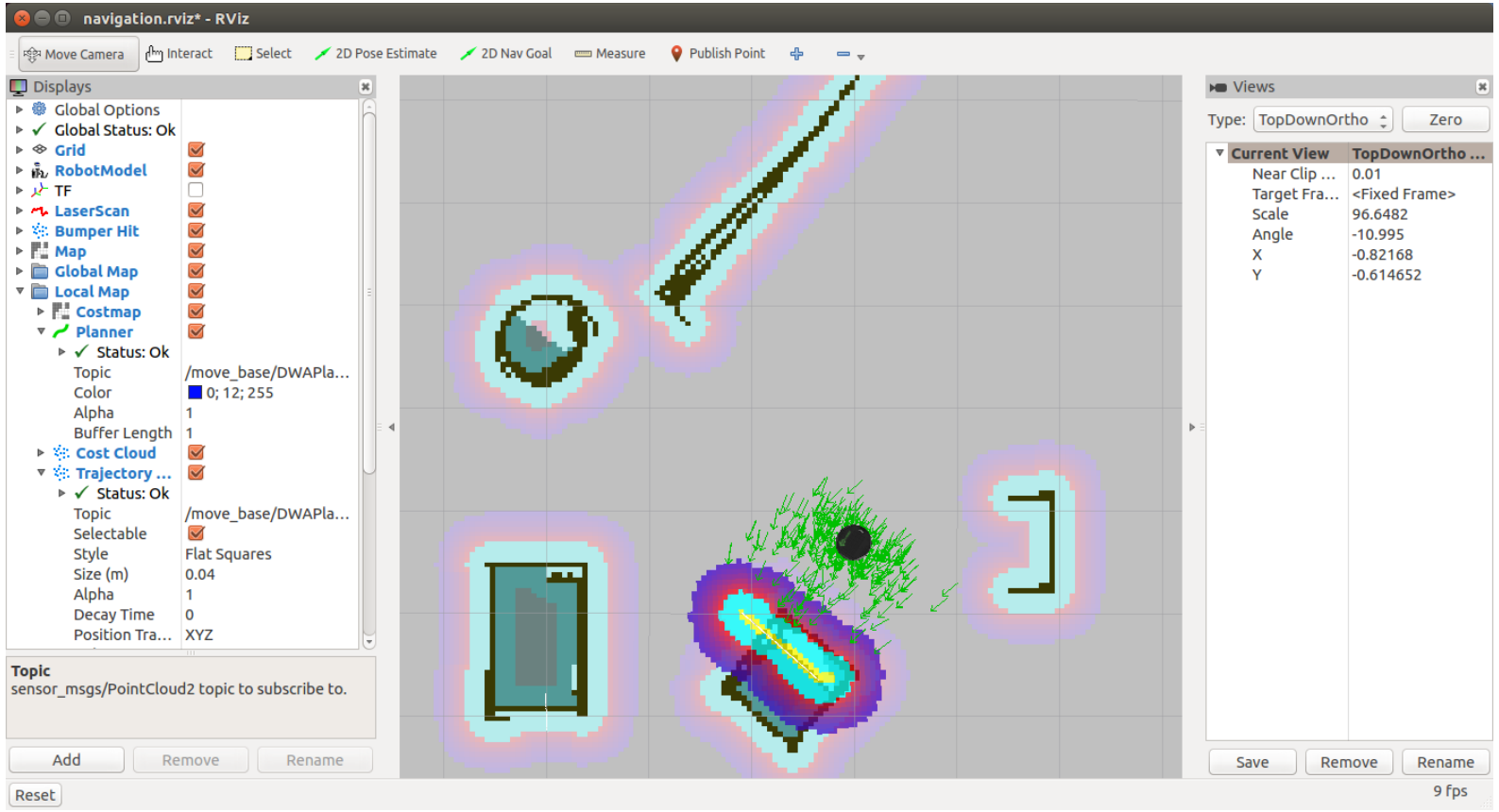


rviz with Navigation Stack

- rviz allows you to:
  - Provide an approximate location of the robot (when starting up, the robot doesn't know where it is)
  - Send goals to the navigation stack
  - Display all the visualization information relevant to the navigation (planned path, costmap, etc.)
- Launch rviz:

```
$ roslaunch turtlebot_rviz_launchers view_navigation.launch
```

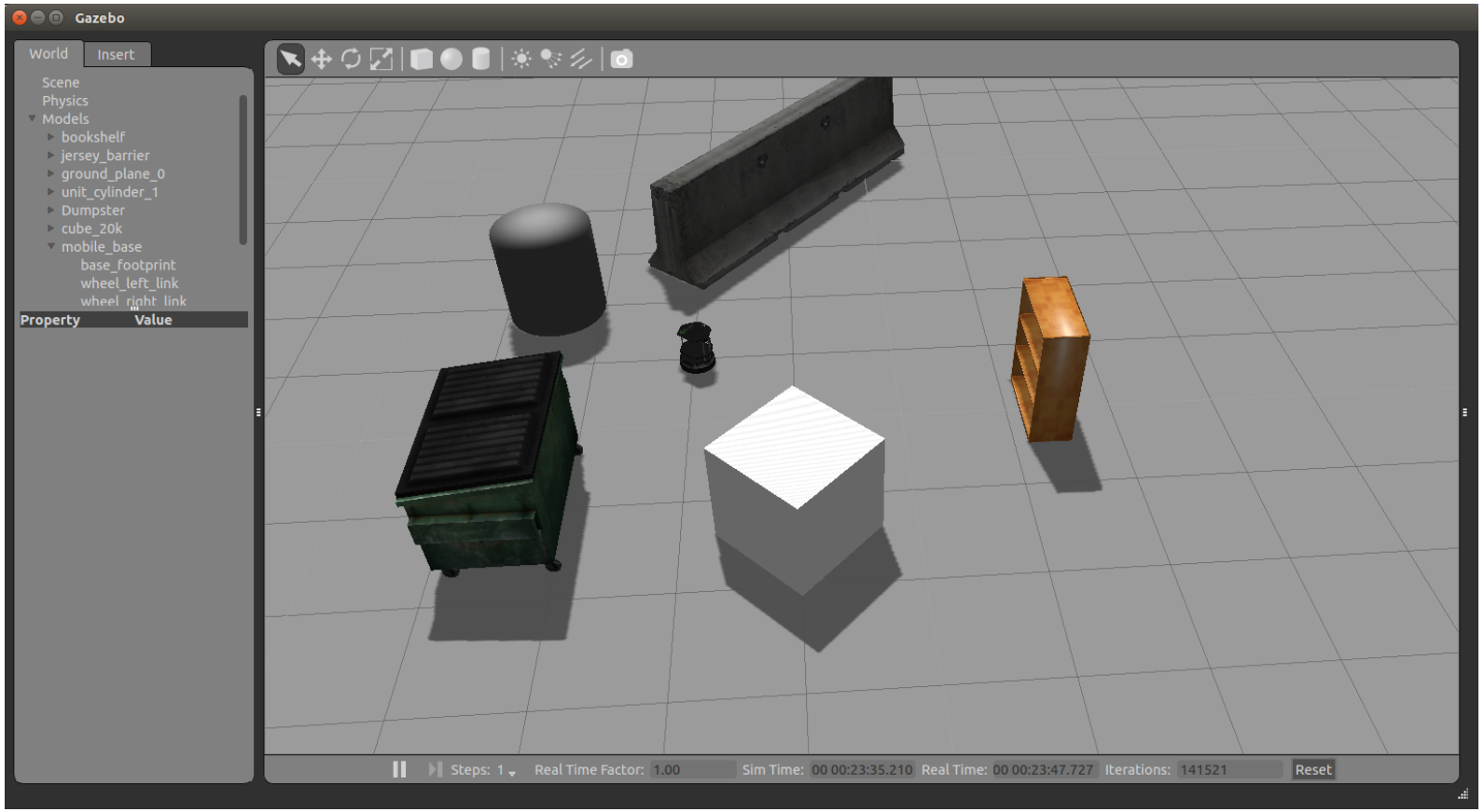
rviz with Navigation Stack



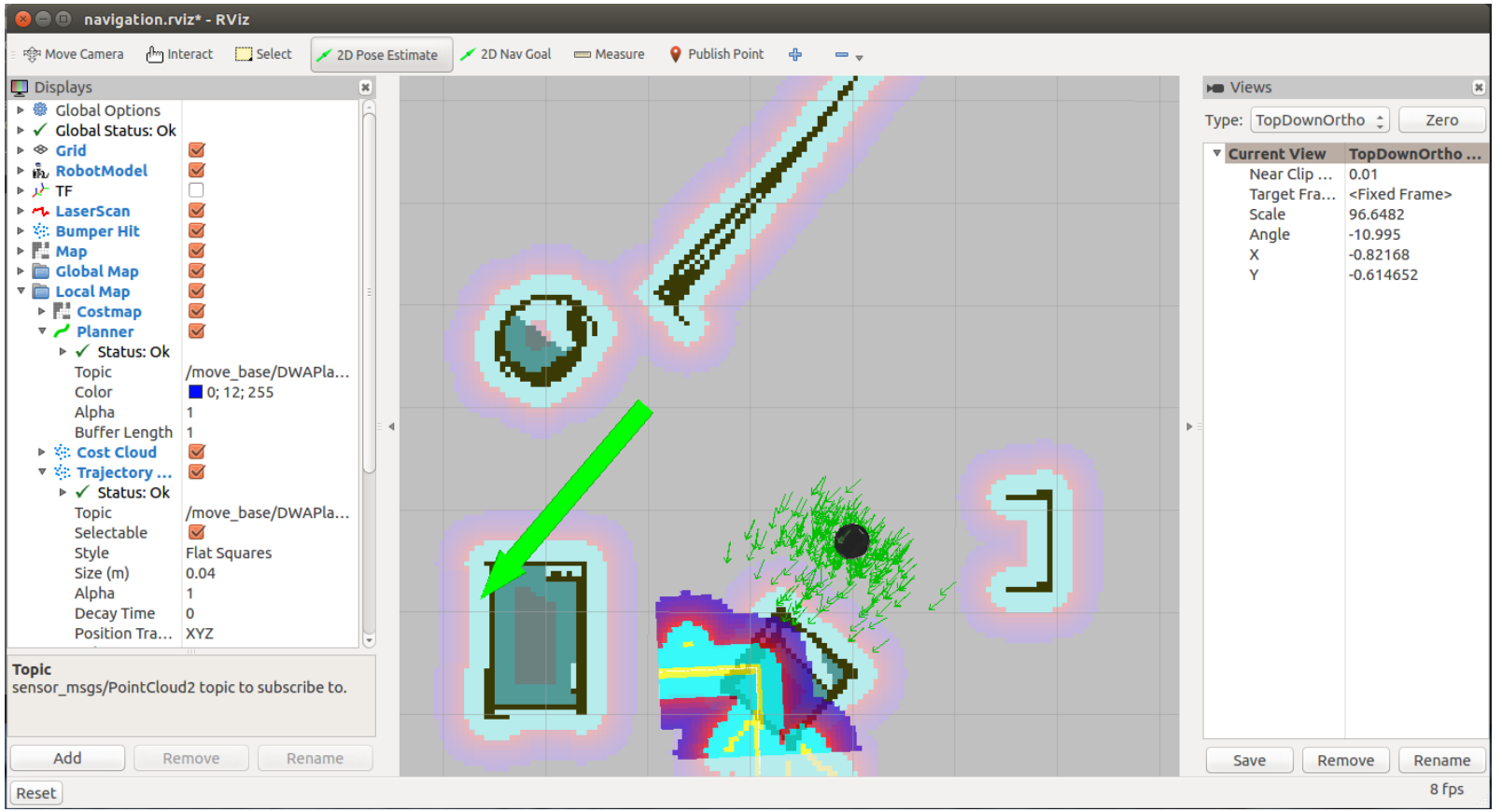
Localize the TurtleBot

- When starting up the TurtleBot doesn't know where it is
- For example, let’s move the robot in Gazebo to (-1,-2)
- Now to provide it its approximate location on the map:
  - Click the "2D Pose Estimate" button
  - Click on the map where the TurtleBot approximately is and drag in the direction the TurtleBot is pointing
- You will see a collection of arrows which are hypotheses of the position of the TurtleBot
- The laser scan should line up approximately with the walls in the map
  - If things don't line up well you can repeat the procedure

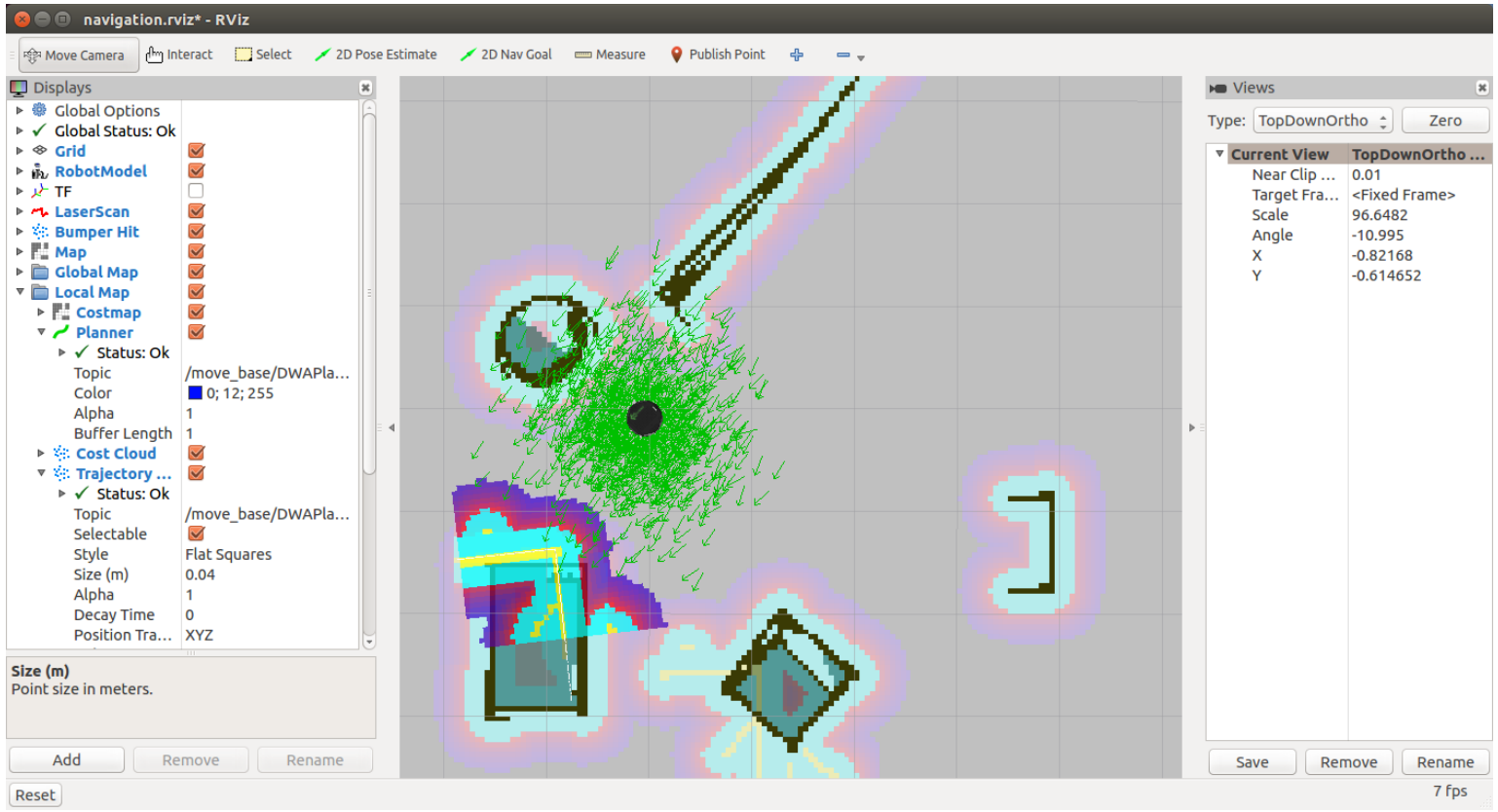
Localize the TurtleBot



Localize the TurtleBot

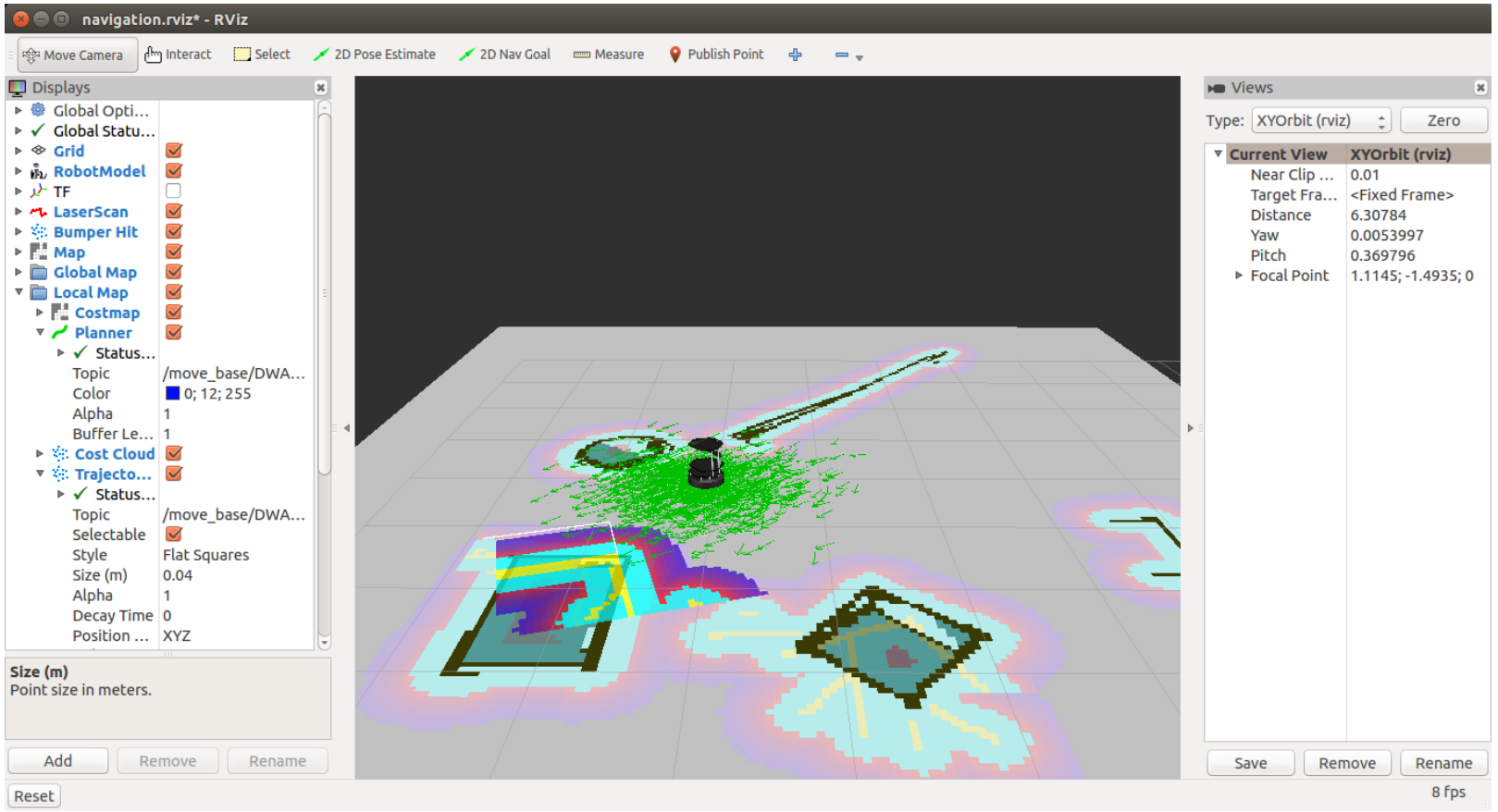


Localize the TurtleBot



Localize the TurtleBot

- You can change the current view (on right panel):



Particle Cloud in rviz

- The **Particle Cloud** display shows the particle cloud used by the robot's localization system
- The spread of the cloud represents the localization system's uncertainty about the robot's pose
- As the robot moves about the environment, this cloud should shrink in size as additional scan data allows amcl to refine its estimate of the robot's position and orientation
- To watch the particle cloud in rviz:
  - Click Add Display and choose Pose Array
  - Set topic name to /particlecloud

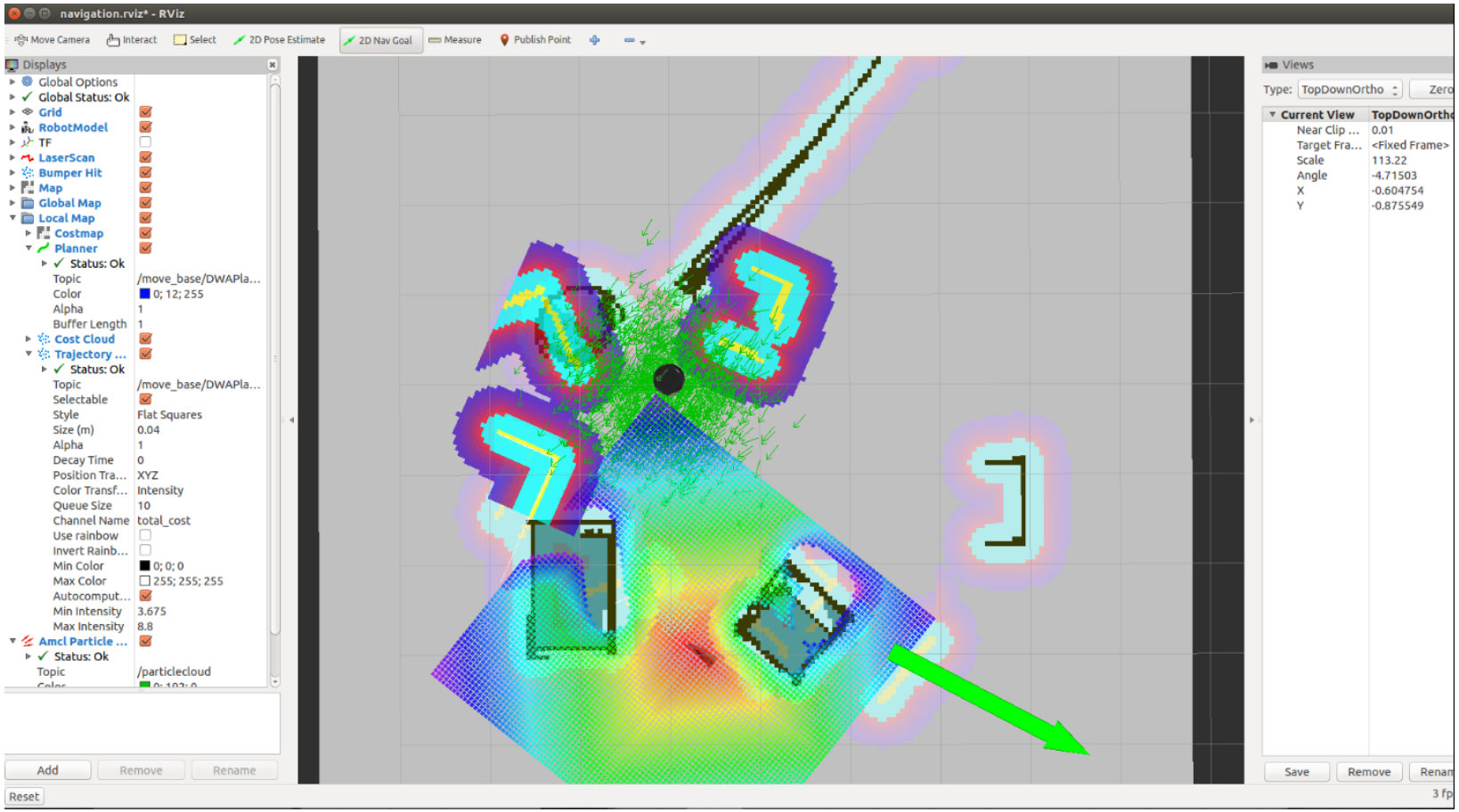
Teleoperation

- The teleoperation can be run simultaneously with the navigation stack
- It will override the autonomous behavior if commands are being sent
- It is often a good idea to teleoperate the robot after seeding the localization to make sure it converges to a good estimate of the position

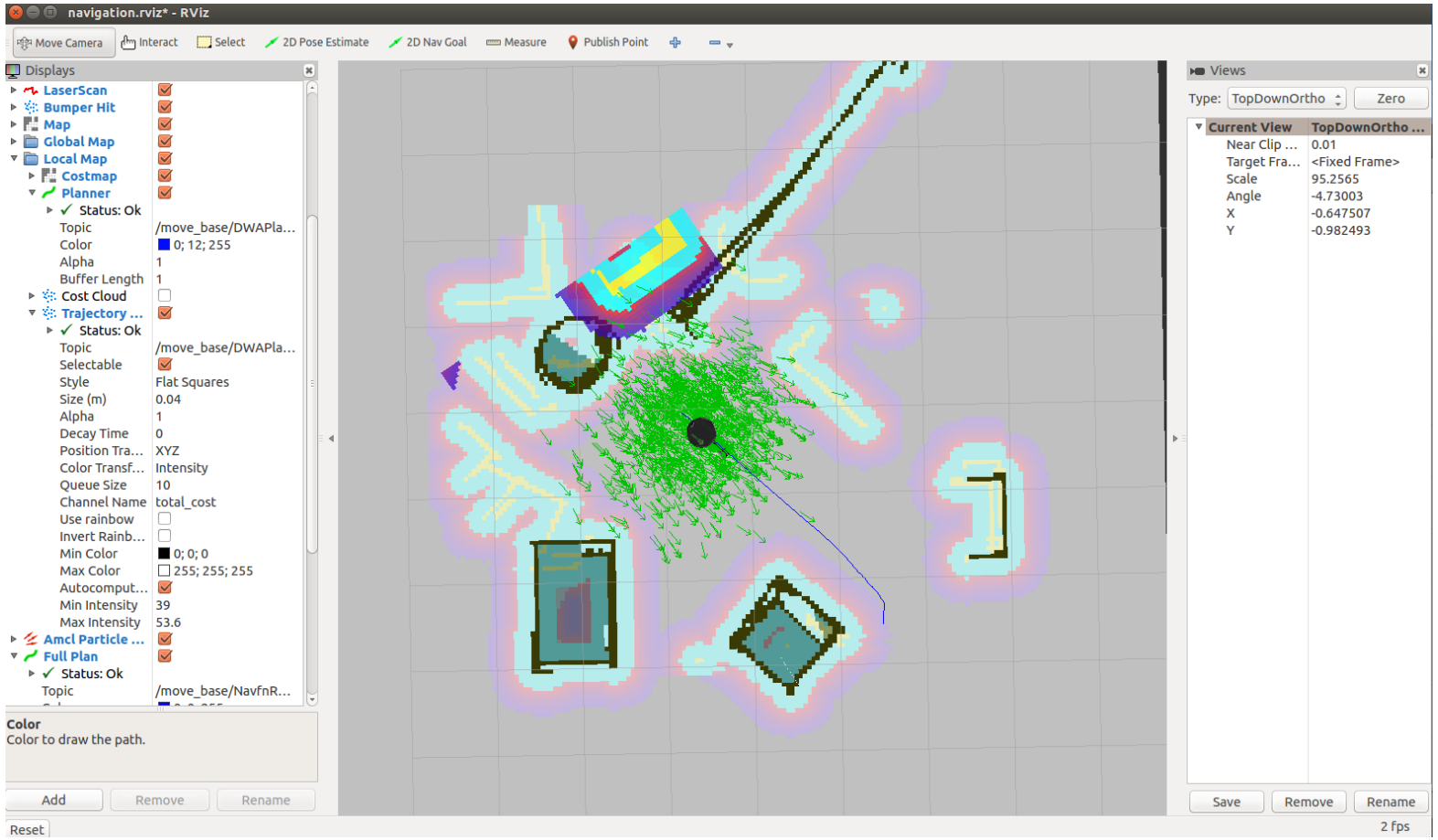
Send a Navigation Goal

- With the TurtleBot localized, it can then autonomously plan through the environment
- To send a goal:
  - Click the "2D Nav Goal" button
  - Click on the map where you want the TurtleBot to drive and drag in the direction where it should be pointing at the end
- If you want to stop the robot before it reaches it's goal, send it a goal at it's current location

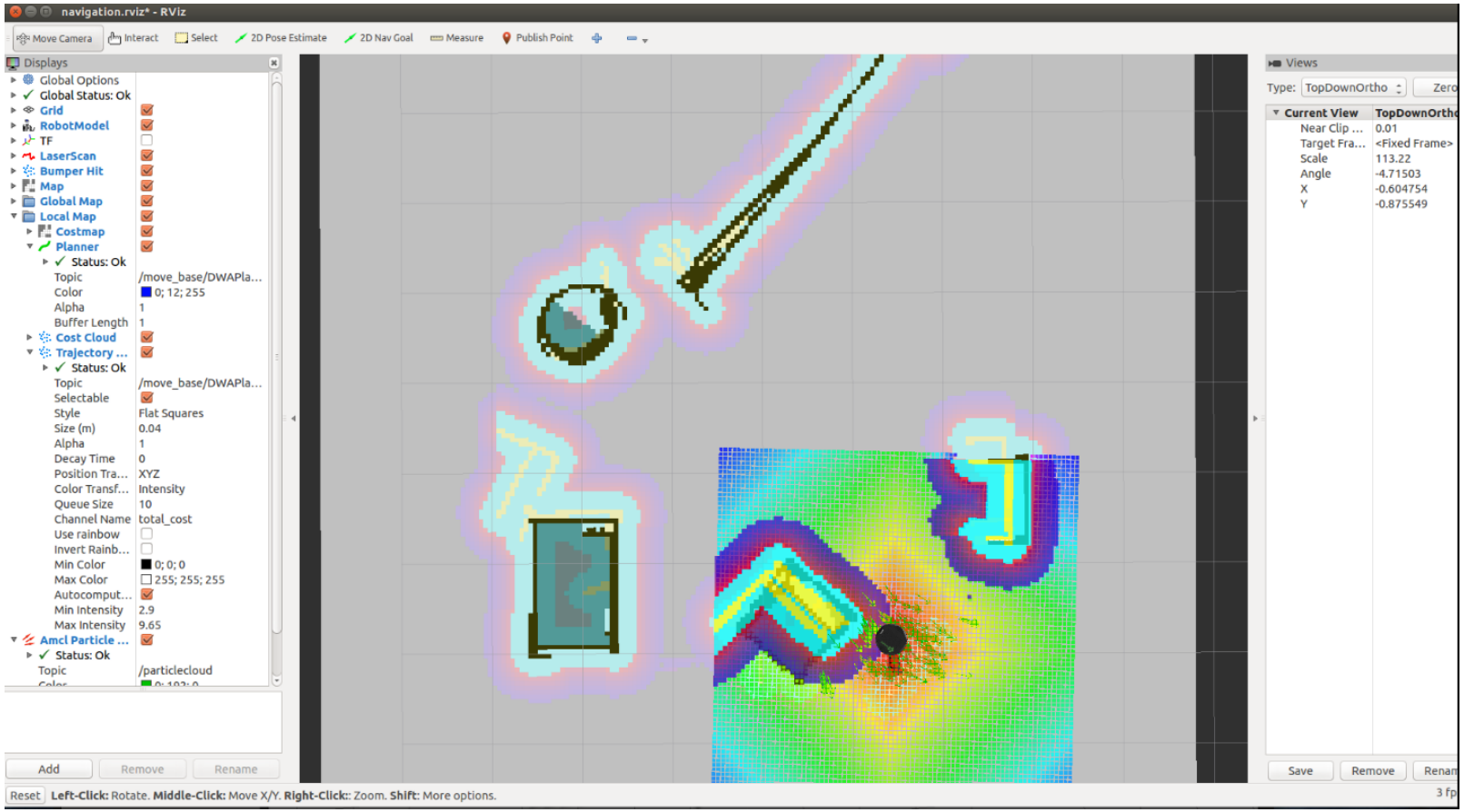
Send a Navigation Goal



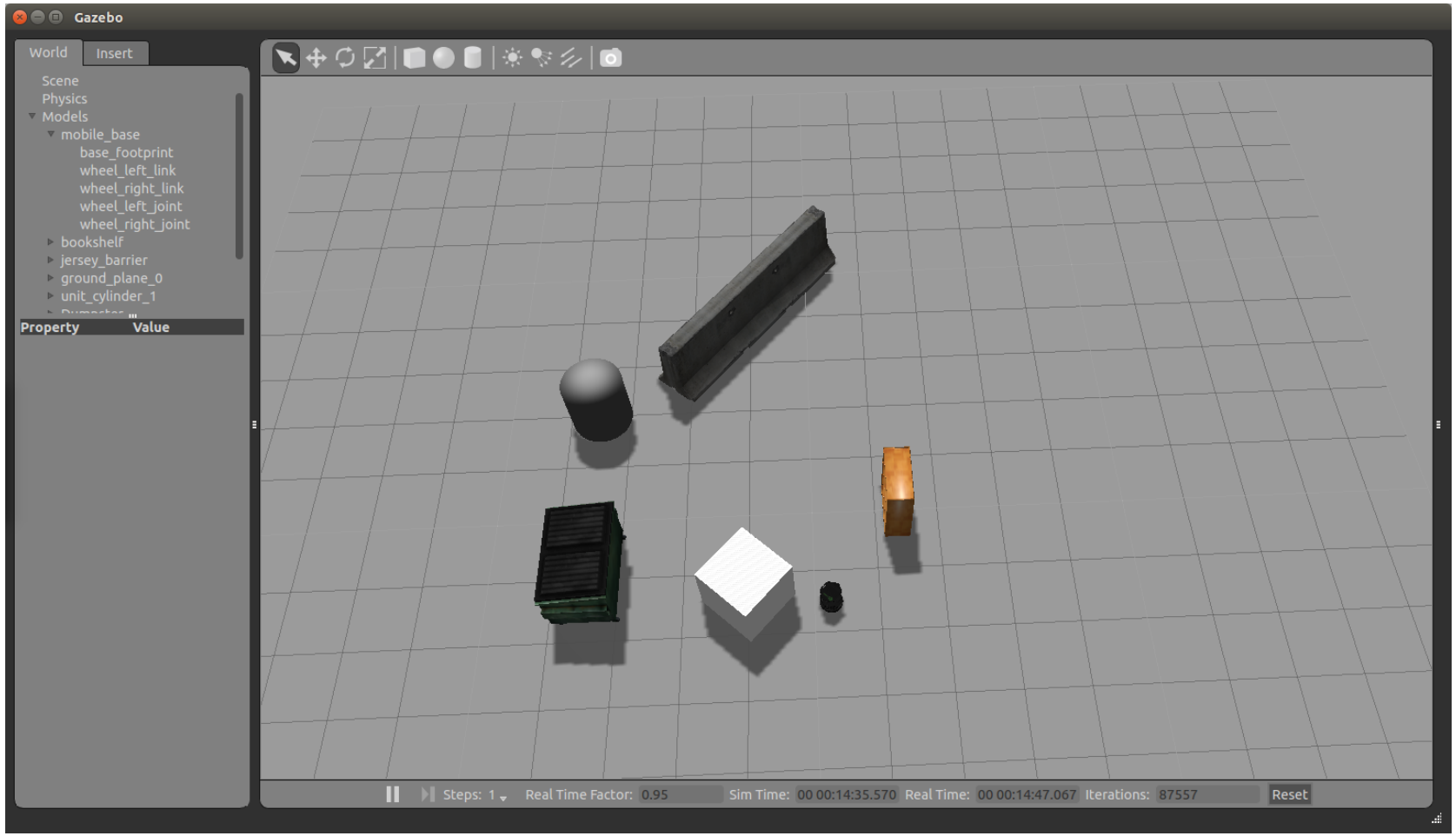
Robot Moves to Destination



Final Pose


navigation.rviz* - RViz
Move Camera
Interact
Select
2D Pose Estimate
2D Nav Goal
Measure
Publish Point
Displays
Global Options
Global Status: Ok
Grid
RobotModel
TF
LaserScan
Bumper Hit
Map
Global Map
Local Map
Costmap
Planner
Status: Ok
Topic /move_base/DWAPla...
Color 0; 12; 255
Alpha 1
Buffer Length 1
Cost Cloud
Trajectory ...
Status: Ok
Topic /move_base/DWAPla...
Selectable
Style Flat Squares
Size (m) 0.04
Alpha 1
Decay Time 0
Position Tra... XYZ
Color Transf... Intensity
Queue Size 10
Channel Name total_cost
Use rainbow
Invert Rainb...
Min Color 0; 0; 0
Max Color 255; 255; 255
Autocomput...
Min Intensity 2.9
Max Intensity 9.65
Amcl Particle ...
Status: Ok
Topic /particlecloud
Add
Remove
Rename
Views
Type: TopDownOrtho
Current View TopDownOrtho
Near Clip ... 0.01
Target Fra... <Fixed Frame>
Scale 113.22
Angle -4.71503
X -0.604754
Y -0.875549
Save
Remove
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Zoom. Shift: More options.

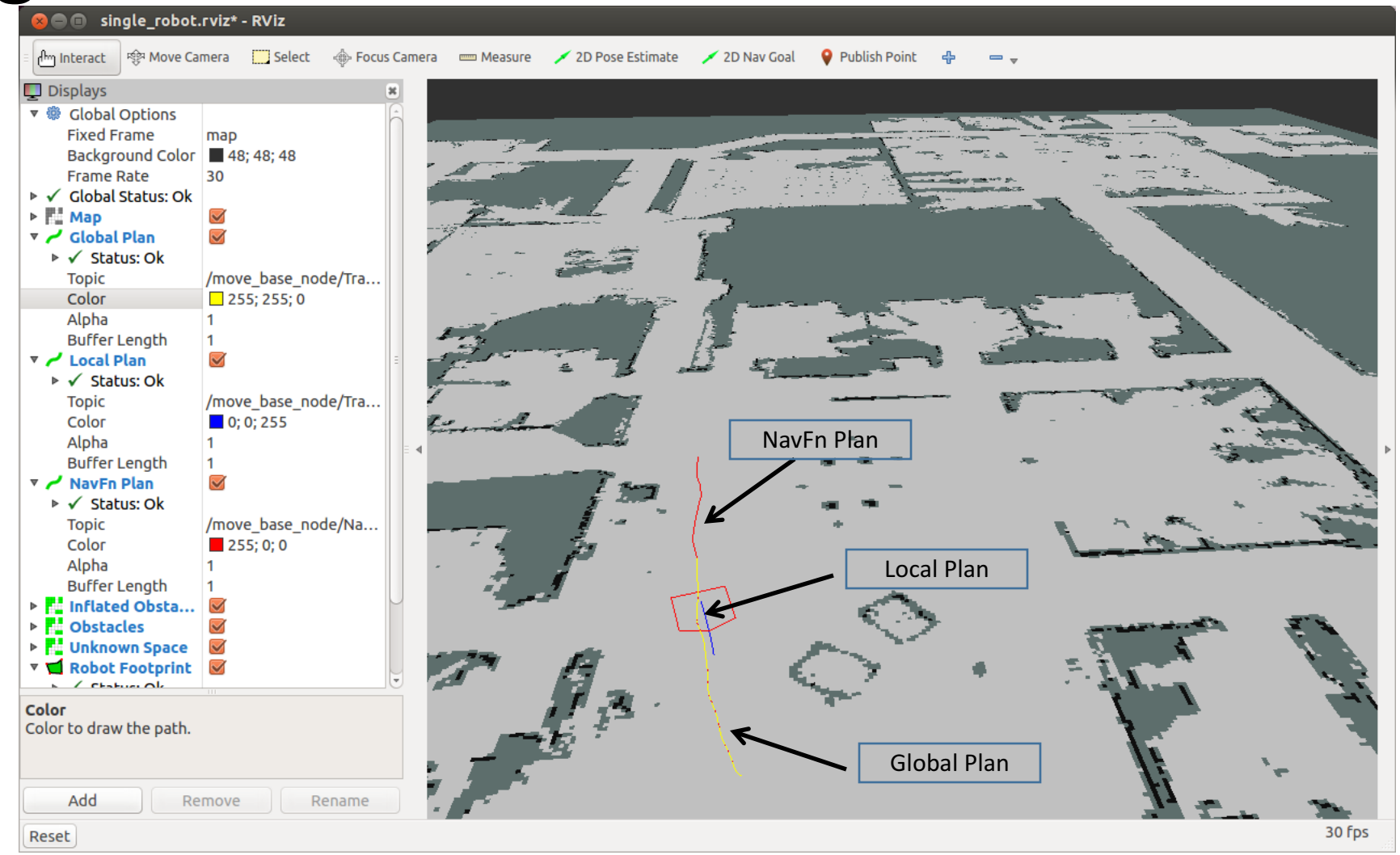
Final Pose In Gazebo



Navigation Plans in rviz

- **NavFn Plan**
  - Displays the full plan for the robot computed by the global planner
  - Topic: /move_base_node/NavfnROS/plan
- **Global Plan**
  - Shows the portion of the global plan that the local planner is currently pursuing
  - Topic: /move_base_node/TrajectoryPlannerROS/global_plan
- **Local Plan**
  - Shows the trajectory associated with the velocity commands currently being commanded to the base by the local planner
  - Topic: /move_base_node/TrajectoryPlannerROS/local_plan

Navigation Plans in rviz



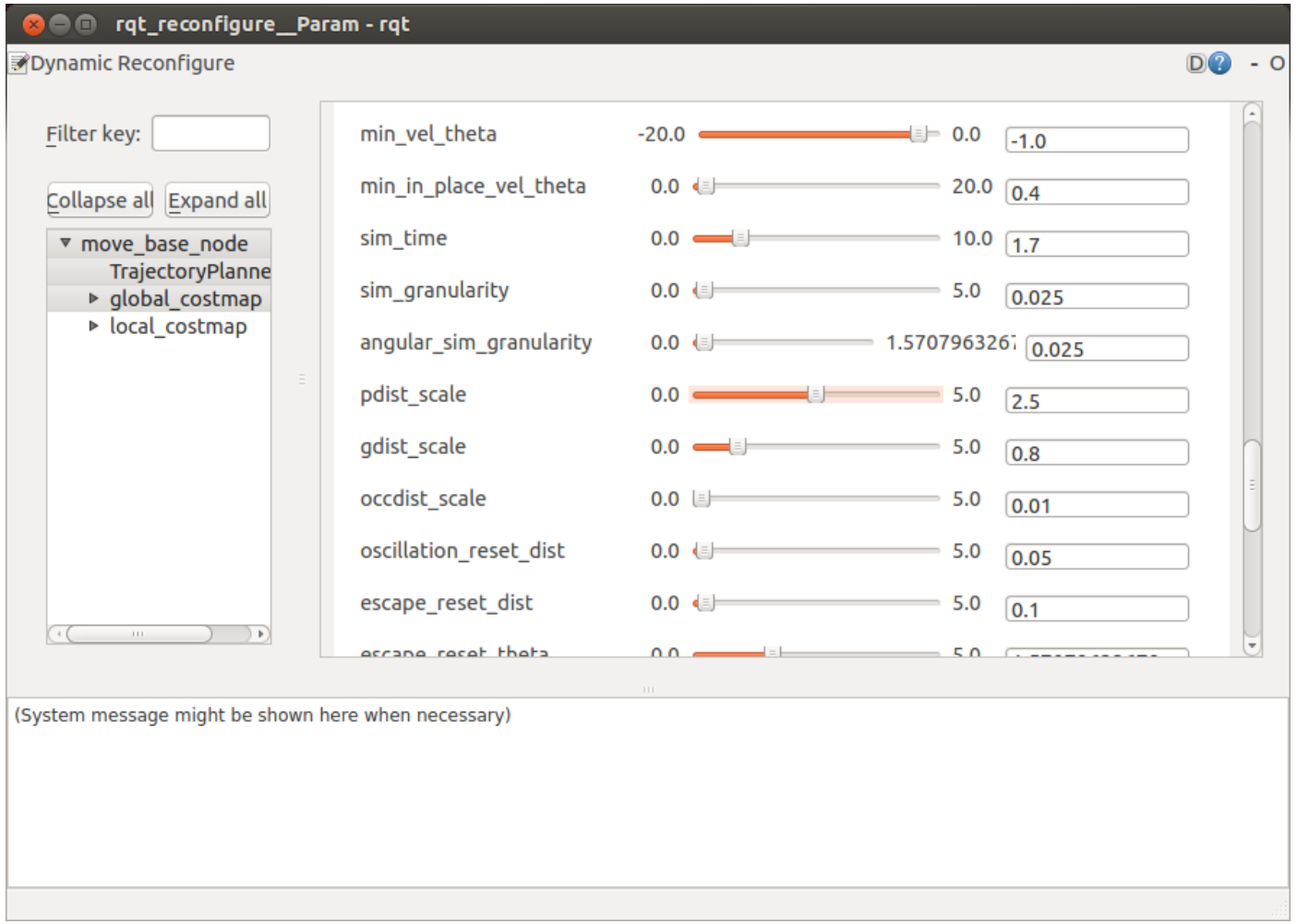
rqt_reconfigure

- A tool for changing dynamic configuration values
- To laun

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

- The navigation stack parameters will appear under move_base_node

rqt_reconfigure



Ex. 7

- Implement a simple navigation algorithm (based on A*) that will use the grid representation of the map to compute a route for the robot from its current location to a given goal location