
CSCE574 – Robotics

Spring 2012 – Notes on Simulating the Turtlebot

This document has some instructions on simulating the Turtlebot robot. It should be helpful in completing Project 2.

1 About the Turtlebot

The remaining projects in this course will be based on a hardware platform called the Turtlebot, which is built from several off-the-shelf components:

1. An iRobot Create mobile base.
2. A small laptop.
3. A Kinect sensor that collects Red-Green-Blue-Depth (RGBD) images.
4. A custom circuit board with both single-axis gyroscope and a power supply for the Kinect.

Note, however, that the first project and many of the tutorials used a very simple robot “simulator” called `turtlesim`, which is designed to help people learn about ROS without worrying too much about the details of real robots. We will not use `turtlesim` again this semester. The fact that both of these names contain the word “turtle” is an unfortunate coincidence.

References

- <http://www.willowgarage.com/turtlebot>
- <http://www.irobot.com/create>
- <http://www.xbox.com/kinect>

2 Installing the simulator

If you are working on the lab computers, the ROS packages you need for this project are already installed, and you can safely ignore the rest of this section.

However, if you would like to use your own Linux computer, you’ll need to install two additional packages, using commands something like this:

```
sudo apt-get install ros-electric-turtlebot-desktop
sudo apt-get install ros-electric-turtlebot-simulator
```

The first package contains the standard tools for interacting with the Turtlebot. These tools work with both the simulated and real Turtlebots. The second package contains the software needed to run the simulation.

However, it turns out that with certain video cards, the simulator runs slowly, unreliably, or (worst of all) produces incorrect results. Perhaps surprisingly, the simulator uses your video card not just to display things onscreen, but also to simulate the data collected by the Kinect sensor. The system requirements link below has some insights into which cards are known to work.

References

- http://ros.org/wiki/turtlebot_simulator/Tutorials/InstallingTurtleBotSimulator
- http://www.ros.org/wiki/simulator_gazebo/SystemRequirements

3 Simulated time vs. wall clock time

In most situations, the ROS library will use the computer's clock to keep track of how much time has passed. This is usually called "wall clock" time.

However, when you are simulating a robot, you might instead want your program to *use the simulator's idea of how much time has passed*, instead of the computer's real clock. This would allow you, for example, to pause, speed up, or slow down the passage of time in the simulator, and still have your programs work correctly. In ROS, this way of measuring time is called "sim time."

To activate sim time, set the parameter `/use_sim_time` to `true`. The launch file is generally the best place for this:

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

If this parameter is set, then everything time-related in ROS—including, for example, the `ros::Rate` class shown in the tutorials, and that many of you have used already—will automatically use sim time instead of wall clock time.

You might be curious about how sim time works: How does your node know how much time is passing in the simulator? The details appear in the link below, but the short version is that the simulator publishes messages on a topic called `/clock`, to which any node using sim time automatically subscribes, behind the scenes.

References

- <http://www.ros.org/wiki/Clock>

4 Starting the simulator

The software usually used to simulate real robots in ROS is called Gazebo.

4.1 Launching Gazebo

To use Gazebo in your project, you'll need to start it as a ROS node by including something like this in your launch file:

```
<node name="gazebo" pkg="gazebo" type="gazebo"
  args="$(find yourPackageName)/worlds/worldFile.world"
  respawn="false"
/>
```

The main thing that might seem unusual about this is the `args` part. Its purpose is pass commandline arguments along to Gazebo, telling it which “world” to simulate. This world file, which should be in a directory called `worlds` under your package, can contain many different details about how the simulation should work. The Gazebo world file format very powerful, but for this course I’ll provide the world files you need, and I won’t ask you make anything more than very simple modifications to them.

4.2 Helping Gazebo find its media

One detail about Gazebo’s world files that is important here is that they sometimes refer to image files to use as maps, backgrounds, textures, or other things. Generally, Gazebo will look in its own standard library for these files, but we will also want to include our own, especially as maps. To accomplish this, we need to add some parameters to the `manifest.xml` for your package:

```
<depend package="gazebo" />
<export>
  <gazebo gazebo_media_path="${prefix}" />
</export>
```

The first line, similar to what you’ve seen before, establishes a dependency between your package and Gazebo. The rest of the snippet passes a parameter to Gazebo, telling it to look in your package’s directory (referred to here by its alias, `/${prefix}`) for any files that the world file asks for. More precisely, Gazebo will look in a subdirectory called `Media` (note the capital ‘M’) within your project for any files that are referenced by the world file.

4.3 Interacting with the Gazebo GUI

You’ll notice that Gazebo creates a window to display the current state of the simulation. This window provides several useful controls.

- Click the pause or play buttons to temporarily interrupt or suspend the simulation. When the simulation is paused, no sim time (see Section 3) will pass.
- You could click the other toolbar icons to add cube-, sphere-, and cylinder- shaped obstacles or light sources of various types. (However, we’ll include all of the obstacles we need in the world file itself, so those tools aren’t needed for this class.)
- Using the File menu, you can reset the simulator to its starting state. Keep in mind that this only resets the simulator, not any of the other ROS nodes.

To help you look at the parts of the simulation that you’re interested in, you can also drag inside the visualization itself.

- Hold down the *left mouse button* and drag to *tilt* your view of the world.
- Hold down the *middle mouse button* and drag up or down to *zoom* in or out.
- Hold down the *right mouse button* and drag to *translate* through the world.

4.4 Controlling the simulation speed

At the bottom of the Gazebo GUI, you'll see several boxes that give information about how time is passing, including timers of how much wall clock time and how much sim time have passed since Gazebo started.

One piece of data that will be of interest to you is the number to the left of the label "× Real Time". If this number is less than 1, then Gazebo's sim time is advancing slower than real time; if this number is greater than 1, then sim time is moving faster than real time.

If Gazebo is too slow, there are at least two things you can try to increase its speed.

- You can increase the length of the time steps used in the simulation. This means that Gazebo (or, more directly, the dynamics library called ODE that Gazebo uses to simulate the physics of its world) will consume less time simulating each second, but may also decrease the accuracy of your robot's simulated motions. To make this change, edit your world file, and increase the value between the `stepTime` tags under `<physics:ode>`.
- You can run Gazebo without its GUI. This means that the computer won't be spending time rendering the visualization that you usually see on the screen. The simulation will run normally in the background, but you won't get a Gazebo window to see its progress. To make this change, add `-g` to the end of the `args` attribute in Gazebo's node tag.

If Gazebo is too fast, you can request that it "throttle" itself by editing the world file and adding this line to the `<physics:ode>` block:

```
<updateRate>-1</updateRate>
```

References

- <http://gazebosim.org/>
- http://gazebosim.org/wiki/sdf_format
- http://www.ros.org/wiki/simulator_gazebo/GazeboConfiguration
- <http://ros.org/wiki/gazebo>
- <http://playerstage.sourceforge.net/wiki/GazeboProblemResolutionGuide>

5 Inserting the TurtleBot into the simulation

The previous section described how to start Gazebo with a specific world, but in ROS, those world files typically do not include the robots themselves.

5.1 Starting the simulated robot

To insert the robot into the simulation, include this snippet in your launch file:

```
<include file="$(find turtlebot_gazebo)/launch/robot.launch"/>
<node pkg="nodelet" type="nodelet" name="openni_manager"
  args="manager" />
```

As you might expect, the result of the first line is to import a collection of launch commands from another, pre-defined launch file. (If you'd like to look at the included launch file, it's in the `turtlebot_gazebo` package.) This sub-launch file includes a node that inserts a simulated Turtlebot into Gazebo, along with other nodes that publish information about the robot's state. (The second line, which likely seems rather mysterious, is to correct an apparent bug in the `robot.launch` file.)

5.2 Analyzing the ROS graph

If you run `rxgraph` at this point, you'll see that the Gazebo node is now publishing over a dozen topics, most of which contain sensor information from the simulated robot. Gazebo also subscribes to a topic called `cmd_vel` that accepts movement commands (See Section 7.3) and to a few other topics that allow some intervention into the simulation.

You should also see a few other nodes designed to provide some higher-level information about what's going on with the robot.

- A node called `robot_pose_ekf`, whose job is to collect information from the robot's encoders (published on `/odom`) and use those data to keep an estimate of the robot's pose, along with measures of how accurate that pose estimate is. This node uses an algorithm called the *Extended Kalman Filter* (EKF) that we will discuss later in the semester. The results are published as a coordinate frame named `/odom_combined` on the topic `/tf`. ROS's tf system for working with coordinations frames and transforms between them is described in more detail in Section 9.
- A node called `robot_state_publisher` whose job is to publish transforms that describe the poses of the robots individual components (sensors, wheels, plates, standoffs, ...) to topic `/tf`.
- A group of nodes called `openni_manager`, `pointcloud_throttle`, `kinect_laser`, and `kinect_laser_narrow`. These nodes work together to convert the depth images returned by the simulated Kinect sensor into messages that look like a scan from a single-plane laser range finder, similar to the laser demo from class. This feature is useful because some other parts of ROS subscribe to these kinds of laser scans, but do not understand the Kinect's raw "point cloud" messages.
- A node called `diagnostic_aggregator`, whose job is to collect, analyze, and categorize diagnostic data from various parts of the robot. It periodically publishes messages on `/diagnostic_agg` to let you know of any (potential) hardware problems. In the simulated version of the robot, there are no diagnostics and no hardware failures, so these messages will simply tell you that the information is "stale."

References

- http://www.ros.org/wiki/robot_state_publisher
- http://en.wikipedia.org/wiki/Extended_Kalman_filter

6 De-cluttering rxgraph

By this point you might be frustrated with the visualization provided by `rxgraph`, because it can become cluttered with lots of links that aren't particularly helpful, such as the subscriptions to the `/clock` and `/rosout` topics. To clear things up a bit, you can use the "Quiet" check box near the top of `rxgraph` to hide a number of nodes that are considered "annoying."

7 Commanding the Turtlebot

Now that you've got a simulation of the TurtleBot running, how can you get the robot to move? This turns out to be quite similar to the procedure we used for the turtlesim turtles.

7.1 Teleoperating the TurtleBot

If you want to move the robot using arrow keys, use this command:

```
roslaunch turtlebot_teleop turtlebot_teleop_key
```

If you include this node in your launch file, be sure to include `output="screen"` in the node tag to give the node access to your terminal.

7.2 (Not) Teleoperating the TurtleBot using Interactive Markers

You may encounter some documentation that describes a technique called “Interactive Markers” to allow you to teleoperate the TurtleBot from within RViz. (We'll get to rviz, which is a GUI for visualizing ROS data, in Section 8.) Unfortunately, this appears to be broken in the version of ROS that's installed in the labs, which is the most recent one available for easy Linux installation. I'm mentioning this here so that you don't waste time trying to get the interactive markers working.

7.3 Moving the TurtleBot from code

While Gazebo is running, you'll notice that it subscribes to a topic called `cmd_vel` of type `geometry_msgs/Twist`, that works in a manner quite similar to the `command_velocity` topic that we worked with in turtlesim.

The details of this message type are:

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

The word “twist” here is, in simple terms, referring to combined linear and angular velocities. To command the robot to move forward, we might send a message like this:

```
geometry_msgs::Twist msg;
msg.linear.x = 0.5;
msg.linear.y = 0.0;
msg.linear.z = 0.0;
msg.angular.x = 0.0;
msg.angular.y = 0.0;
msg.angular.z = 0.0;
```

To command the robot to rotate in place, we might send a command like this:

```
geometry_msgs::Twist msg;
msg.linear.x = 0.0;
msg.linear.y = 0.0;
msg.linear.z = 0.0;
msg.angular.x = 0.0;
msg.angular.y = 0.0;
msg.angular.z = 0.5;
```

You should already be familiar with the details of how to create a publisher for this topic, and how to actually send the message. If you need a refresher, please refer to the previous document called “Notes on ROS” document on the course website.

8 Visualizing the robot and its data with RViz

So far we’ve talked about how to simulate the TurtleBot, and how to command that simulated robot to move. In addition to these things, ROS also provides a tool called RViz that can display a wide variety of information about how the robot itself is operating.

To start RViz, use the command

```
roslaunch rviz rviz
```

or include the corresponding line in your launch file.

8.1 RViz is not Gazebo

You might notice that the windows created by RViz and Gazebo look a bit similar at first glance. It is important to understand the differences between these two programs. Gazebo is a robot simulator, and its window shows a view of *what is actually happening in the simulated world*. In contrast, RViz is a *visualization tool for data available to the robot*. Gazebo shows where the simulated obstacles “really are”; RViz can show only the obstacles specifically sensed and mapped out by the robot.

In particular, keep in mind that when you are running the real robots later in the semester, you won’t be using Gazebo at all. RViz, however, is perfectly happy to show data from the real robot or from Gazebo.

8.2 Setting the fixed frame

We’ll display some data shortly, but first we need to tell RViz what coordinate system to use. Basically, we need to choose one object in the world to hold fixed as the “origin” of the visualization. In ROS terms, we need to choose a *frame* (see Section 9) to act as a *fixed frame*. To choose the fixed frame, use the drop-down box on the left under “Global Options.” For the TurtleBot, there are at least two natural choices for the fixed frame:

- The location on the floor directly under the center of the robot is represented by a frame called `/base_footprint`. Choosing this for the fixed frame will put the robot at the origin. Then as the robot moves, we’ll see everything else in the world move around it.

-
- The movements estimated by the robot's EKF (see Section 5.2) are published as a frame called `/odom_combined`. Choosing this for the fixed frame provides more of a "global view" in which we can see the robot moving through the world.

8.3 Working with RViz displays

When you start RViz for the first time, most of its window will be taken up with a black rectangle. This is the 3D view of the world; nothing is displayed there yet because we haven't asked RViz to display any data.

RViz refers to the different kinds of data that we might want to show as *displays*. To add a new display, use the "Add" button near the bottom left of the window, then select the type of display that you want. After adding the display in this way, it will show up in the Displays panel on the left, and generally will need to be configured there.

There are several different displays that you may want to add when you're working with the TurtleBot.

- A *Grid* display as a reference point for where the ground is.
- A *Robot Model* display to show the robot itself. You might be curious about how RViz knows what our robot looks like. It reads the robot's physical description as a (lengthy) string from a ROS parameter called `robot_description`. This robot description is in a format called Unified Robot Description Format (URDF). In our setup, the `robot_description` parameter is set in the `robot.launch` portion of our launch configuration. This arrangement guarantees that Gazebo and RViz are using the same robot model.
- A *Camera* display to show the images that the robot's camera is receiving. After adding this display, you'll need to configure the topic on which it listens to `/camera/image_raw`.
- A *Point Cloud 2* to show the raw depth data collected by the Kinect sensor. After adding this display, you'll need to set the topic to `/camera/depth/points`. You may also want to experiment with the color settings for this display to make the points easier to see.
- A *Laser Scan* to show the "faked" laser scan data derived from the Kinect. (See Section 5.2.) Don't forget to set the topic name, for which you'll have two choices.

All of these together provide a wealth of information. A reasonable practice is to include all of them in your RViz configuration, but to use the check boxes next to each display's name to hide the information that you are not interested at a particular time. Note that some of these displays can be resource hogs, so you may be able to improve performance by selective disabling.

8.4 RViz display configurations

If you've started RViz several times, you might have noticed that it remembers your displays and their configuration from the previous run. RViz stores this kind of information in a file called a `vcg` (Visualization Configuration) file. There are several things you can do with this kind of file to force RViz to behave nicely.

- To save (or load) the RViz configuration to (or from) a `vcg` file, you can use the appropriate commands in the File menu of RViz.
- To force RViz to use a specific, existing `vcg` file when it starts up, you can pass it a `-d` followed by the file name on its command line. In a launch file, you can put the `-d` and filename in the `args` attribute.

-
- When RViz starts without a `-d`, it automatically loads configuration from a file in your home directory called `.rviz/display_config`. If you would like to return to the default, empty configuration, you should close RViz, delete this file, then reopen RViz.

8.5 Segfaults and “Bad Drawable” errors

Depending on the specific hardware on which you run RViz, you may sometimes see errors when you attempt to start RViz. This could be either a dreaded “Segmentation Fault”, or an error about something called a “Bad Drawable”. In both cases, the error is fatal but intermittent. This is a symptom of a known bug when RViz is used with certain video cards. (See Section 2). There are two workarounds you can try.

- First, you can try setting the environment variable `OGRE_RTT_MODE` to one of these three strings: `Copy`, `PBuffer`, or `FBO`. This setting changes some of the internal details of how the window is rendered. Depending on your configuration, you may have better success with one of these options than others.
- Second, since the problem is intermittent, you can simply try starting RViz again and again until it succeeds. The best way to do this is to use a launch file, and include `respawn="true"` in the node tag for RViz.

More details appear in the bug tracker link shown below.

References

- <http://www.ros.org/wiki/rviz>
- <https://code.ros.org/trac/ros-pkg/ticket/4830>

9 TF: Working with Transforms

At several points in the previous sections, we’ve run into the ideas of *coordinate frames* and *transforms* between them. We discussed frames and transforms in class, and you’ll be glad to know that ROS provides a package called “tf” that can make these issues relatively painless, as long as we are careful.

9.1 TF overview

Every time you are working with coordinates, you should know which frame those coordinates are expressed in. (Otherwise, the likelihood of making a mistake is quite high.) In tf, each coordinate frame is identified by short string called a `frame_id`. The idea of tf is to use a topic called `/tf` to publish transforms that describe how to convert coordinates in one frame to another. Each message on this topic contains one or more transforms that look like this:

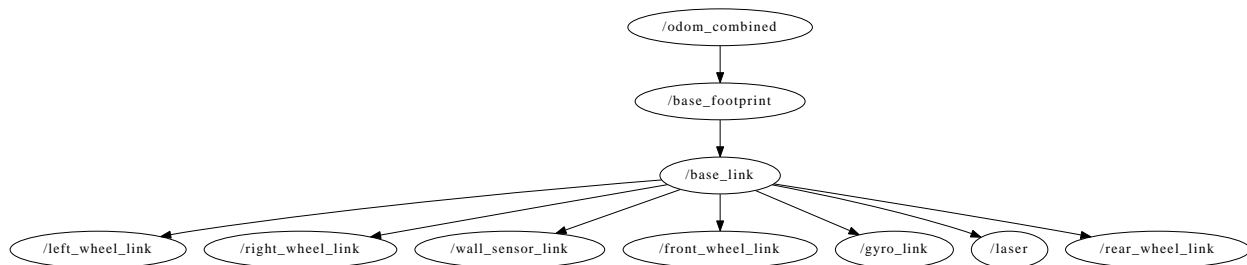
```

Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/Transform transform
  geometry_msgs/Vector3 translation
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion rotation
    float64 x
    float64 y
    float64 z
    float64 w

```

This data type contains two `frame_id` strings: One in the header, called simply `frame_id`, and another called `child_frame_id`. This transform tells us how to convert coordinates expressed in the frame named by `frame_id` into the frame named by `child_frame_id`.

Notice that this setup forms a directed graph, in which the nodes are coordinate frames, and the directed edges are transforms between those frames. In `tf`, each frame can have only one “parent” frame, so we actually get a tree structure. Here’s a portion of the `tf` tree that is used with the Turtlebot:



9.2 Looking up transforms in within your code

In principle, you now know enough to convert between coordinate frames. You could subscribe to `tf`, remember the transforms that are published on that topic, and compose them carefully to compute the correct transform to move points from one frame to another. The good news is that **there is no need to explicitly subscribe to `/tf`, because the ROS client library has C++ classes that handle all of the dirty work for us.**

To do this, there are a few steps.

1. Make sure your package depends on the `tf` package.
2. Create a transform listener.
 - Include `<tf/transform_listener.h>`.
 - Declare a variable `tf::TransformListener` object. This object will subscribe to `tf` and cache the transforms that are broadcast after it is created. Therefore, its is important to
 - (a) create this object after the node is initialized (that is, after the `NodeHandle` is created), and

(b) destroy it only after you are completely finished using the tf system (that is, when your program ends).

For example, if you create a new TransformListener each time you need to change coordinate frames, then its cache will be empty, and it won't be able to return the transformations you need.

- Ask your transform listener for the transform between the two frames you care about.
 - Create an object of type `tf::StampedTransform`:

```
tf::StampedTransform transform;
```

- Inside a try/catch block ask the transform listener for the appropriate transform:

```
try {
    listener.lookupTransform(
        "toFrame",
        "fromFrame",
        ros::Time(0),
        transform
    );
} catch (tf::TransformException ex) {
    ROS_ERROR("%s", ex.what());
}
```

Note that this method takes care of the details of finding a path through the transform tree and composing all of the transforms along the way. For example, in the tree shown above, it would be just fine to ask for the transform between `/left_wheel_link` and `/right_wheel_link`—tf will automatically find a path between these two via `/base_link` and perform the appropriate inversion and composition to return a transform directly between the body frames of the left and right wheels.

The `ros::Time(0)` in the code above means that we want to know the *most recent* transform between those two frames. Time 0 is a special case. A more likely situation is that you'll want to give a specific time stamp to ask for the transform that was correct at some specific point in the past. These timestamps are particularly important when you are processing sensor data. By the time you are processing a sensor data message, the corresponding transforms may already have changed. Therefore, you should use the timestamp of message containing the sensor data instead of `ros::Time(0)`.

The try/catch block is important here, because `lookupTransform` can generate exceptions even if you've done everything right. The most common reason for this to occur is that the transform listener has not yet received enough messages on topic `/tf` to construct the transform you need. These things often resolve themselves with time.

3. Use the transform to change coordinates between your two frames.

- Store the point you care about in a `tf::Point`.

```
tf::Point pointInFromFrame
pointInFromFrame.setX(yourX);
pointInFromFrame.setY(yourY);
pointInFromFrame.setZ(yourZ);
```

- Multiply that point by the transform object and store the resulting `tf::Point`.

```
tf::Point pointInToFrame = transform * pointInFromFrame;
```

-
- Access the coordinates of the transformed point:

```
ROS_INFO("Transformed point is (%f, %f, %f).",
        pointInToFrame.x(),
        pointInToFrame.y(),
        pointInToFrame.z()
    );
```

9.3 Frames in RViz

You may find it helpful to add a tf display to RViz to visualize the coordinate frames are. In this display, the x -, y -, and z -axes of each frame are shown in red, green, and blue respectively. You can reduce the clutter from this display by disabling the “Show Names” option, and by disabling some (or most) of the transforms.

9.4 TF from the command line

In addition to the client library functions described above, tf also includes a couple of command line tools that can help you figure out what’s going on.

First, to see the current tf tree, use a command like this:

```
roslaunch tf view_frames
```

This node will subscribe to /tf for a few seconds, and then create a file called frames.pdf that illustrates all of the frames for which transforms are currently being published. (The image above is a highly simplified version of what you might see in frames.pdf.)

Second, you can see specific the transforms between a pair of frames using the tf_echo command. Here’s an example:

```
roslaunch tf tf_echo /kinect_depth_frame /base_link
```

9.5 The odometric frame

In the context of our Turtlebots, one particular frame deserves special attention, since it’s not a body frame of any specific object. The frame /odom_combined has its origin at the robot’s *starting position*, with its positive x -axis in the direction the robot was facing at the start.

Note, however, that the transforms between this frame and the robot’s body frames are *estimated* (by the EKF mentioned above). Therefore, the robot’s position within this frame can be interpreted as an estimate of how far an in what direction the robot has moved since it started. Over time you should expect this estimate to drift away from the robot’s real pose.

References

- <http://www.ros.org/wiki/tf/Tutorials/Introductiontotf>
- [http://www.ros.org/wiki/tf/Tutorials/Writingatflistener\(C++\)](http://www.ros.org/wiki/tf/Tutorials/Writingatflistener(C++))
- <http://www.ros.org/wiki/tf/Overview/DataTypes>

-
- <http://ros.org/doc/electric/api/tf/html/c++/>
 - <http://www.continuousphysics.com/Bullet/BulletFull/index.html>