Simulation with ROS and Gazebo

Robot Operating System

Robot Operating System (ROS) is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.
http://wiki.ros.org/ROS/Introduction

ROS packages

Packages are the main unit for organizing software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together.
http://wiki.ros.org/Packages

ROS workspaces

To aid in organizing, developing and building ROS packages, they can be placed in special folders called workspaces. With them you can overlay ROS packages installed with the system's package manager. For example, if you want to modify a ROS package, you just have to download its source, place it into a workspace and build it, if you activate the workspace in the terminal, all later programs you run will 'see' the package you built, instead of the system one.
Activating a workspace also configures the terminal autocomplete feature (<Tab>), with the available packages and executables.
http://wiki.ros.org/catkin/workspaces
Commands:

```
cd ~/catkin_ws # Change to the workspace folder
catkin_make # Build it
source devel/setup.bash # Activate it
```

ROS nodes

ROS nodes are executables, that use ROS to communicate. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one Node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provides a graphical view of the system, and so on.

To run a ROS node, a ROS Master node must be running, which handles all running nodes and their communications.

http://wiki.ros.org/Nodes
http://wiki.ros.org/rosnode
Commands:

```
roscore # Run ROS master
rosrun <package> <executable> # Run a ROS node
rosrun teleop_twist_keyboard teleop_twist_keyboard.py # Example
rosnode list # List running ROS nodes
```

ROS topics

Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.

Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching type.

http://wiki.ros.org/Topics

Commands:

```
rostopic list# List topics
rostopic echo /topic_name # Listen to a ROS topic
```

roslaunch

Roslaunch is a tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters. It includes options to automatically respawn processes that have already died. roslaunch takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.

http://wiki.ros.org/roslaunch

Commands:

```
roslaunch package_name file.launch # Run a launch file
```

Gazebo

Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs.

Although not part of ROS, it supports ROS through an interface, through which ROS nodes can access the simulated environment and robots.

http://gazebosim.org/

URDF

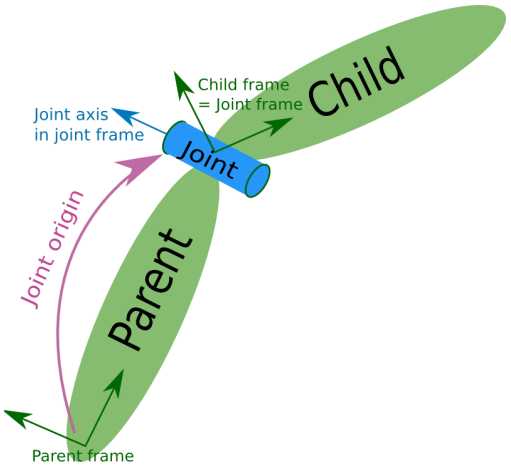
The Unified Robot Description Format (URDF) is an XML specification to describe a robot. Only tree structures can be represented, also the specification assumes the robot consists of rigid links connected by joints.

The description of a robot consists of a set of link elements, and a set of joint elements connecting the links together. So a typical robot description looks something like this:

```
<robot name="pr2">
  <link> ... </link>
  <link> ... </link>
  <link> ... </link>
  <joint>  ....  </joint>
  <joint>  ....  </joint>
  <joint>  ....  </joint>
</robot>
```

http://wiki.ros.org/urdf/XML/model

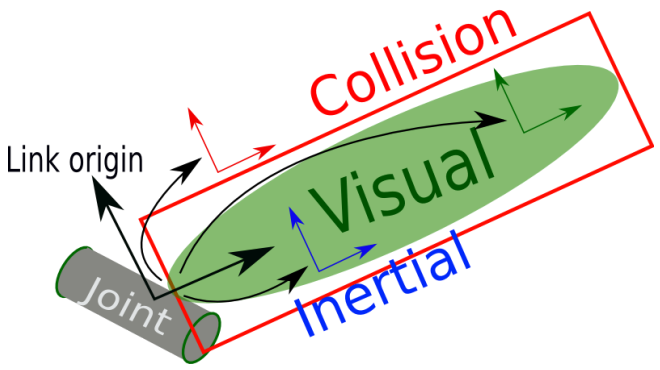
The joint element describes the kinematics and dynamics of the joint and also specifies the safety limits of the joint. During this laboratory only fixed joints are used.




```
<joint name="my_joint" type="fixed">
    <!-- Use xyz to control position and rpy to control orientation
    of the following link -->
    <origin xyz="0 0 1" rpy="0 0 3.1416"/>
    <parent link="link1"/>
    <child link="link2"/>
</joint>
```

http://wiki.ros.org/urdf/XML/joint

The link element describes a rigid body with an inertia, visual features, and collision properties. During this laboratory only box geometries are needed.



```
 <link name="my_link">
    <inertial>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0"
izz="100" />
    </inertial>

    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <box size="1 1 1" />
      </geometry>
    </visual>

    <collision>
      <origin xyz="0 0 0" rpy="0 0 0"/>
      <geometry>
 <!-- The collision geometry should be the same as the visual -->
        <cylinder radius="1" length="0.5"/>
      </geometry>
    </collision>
  </link>
```

http://wiki.ros.org/urdf/XML/link

Xacro

Xacro (XML Macros) Xacro is an XML macro language. With xacro, you can construct shorter and more readable XML files by using macros that expand to larger XML expressions. Mostly used for URDF files.

http://wiki.ros.org/xacro

Gazebo sensor plugins

Gazebo plugins give your URDF models greater functionality and can tie in ROS messages and service calls for sensor output. During the laboratory we will use 3 different kind of sensors: camera, lidar and GPS.

http://gazebosim.org/tutorials?tut=ros_gzplugins

http://wiki.ros.org/hector_gazebo_plugins#GazeboRosGps

Example camera configuration:

```
<!-- camera -->
 <gazebo reference="camera_link">
   <sensor type="camera" name="camera1">
     <update_rate>30.0</update_rate>
     <camera name="head">
       <horizontal_fov>1.3962634</horizontal_fov>
       <image>
         <width>800</width>
         <height>800</height>
         <format>R8G8B8</format>
       </image>
       <clip>
         <near>0.02</near>
         <far>300</far>
       </clip>
       <noise>
         <type>gaussian</type>
         <mean>0.0</mean>
         <stddev>0.007</stddev>
       </noise>
     </camera>
<plugin name="camera_controller" filename="libgazebo_ros_camera.so">
       <alwaysOn>true</alwaysOn>
       <updateRate>0.0</updateRate>
       <cameraName>camera1</cameraName>
       <imageTopicName>image_raw</imageTopicName>
       <cameraInfoTopicName>camera_info</cameraInfoTopicName>
       <frameName>camera1_link</frameName>
     </plugin>
   </sensor>
 </gazebo>
```

Example LIDAR configuration

```
    <gazebo reference="laser">
      <turnGravityOff>true</turnGravityOff>
      <sensor type="ray" name="laser">
        <pose>0 0 0 0 0 0</pose>
        <visualize>false</visualize>
        <update_rate>50</update_rate>
        <ray>
          <scan>
            <horizontal>
              <samples>720</samples>
              <resolution>1</resolution>
     <!-- Change to 360 degree, watch out, config expects radians-->
```

```
              <min_angle>-2.35619</min_angle>
              <max_angle>2.35619</max_angle>
            </horizontal>
          </scan>
          <range>
            <min>0.1</min>
            <max>30.0</max>
            <resolution>0.01</resolution>
          </range>
          <noise>
            <type>gaussian</type>
            <mean>0.0</mean>
            <stddev>0.001</stddev>
          </noise>
        </ray>
        <plugin name="gazebo_ros_laser"
filename="libgazebo_ros_laser.so">
          <topicName>scan</topicName>
          <frameName>laser</frameName>
          <robotNamespace>/</robotNamespace>
        </plugin>
      </sensor>
    </gazebo>
```

Example GPS configuration, it does not need link.

```
<gazebo>
    <plugin name="gps_controller"
filename="libhector_gazebo_ros_gps.so">
      <robotNamespace>/</robotNamespace>
      <updateRate>40</updateRate>
      <bodyName>base_link</bodyName>
      <frameId>base_link</frameId>
      <topicName>navsat/fix</topicName>
      <velocityTopicName>navsat/vel</velocityTopicName>
      <referenceLatitude>50.0</referenceLatitude>
      <referenceLongitude>60.0</referenceLongitude>
      <referenceHeading>0</referenceHeading>
      <referenceAltitude>0</referenceAltitude>
      <drift>0.0001 0.0001 0.0001</drift>
    </plugin>
  </gazebo>
```

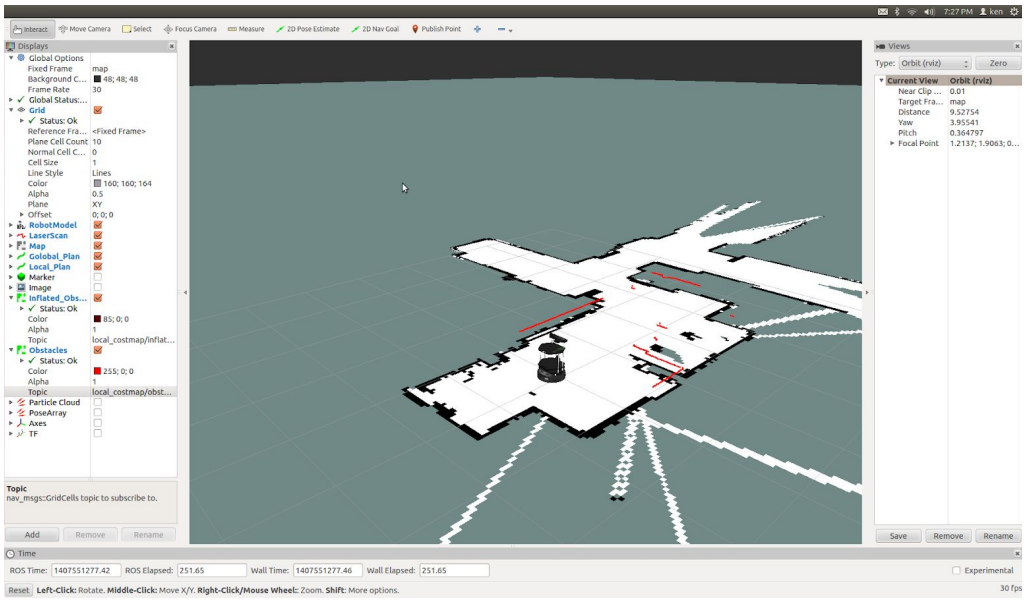
Rviz

Rviz is a 3D visualization tool for ROS. It can visualize RobotModels, LaserScans and through interactive markers it can control robots.

At startup, it is necessary to set the fixed frame, in the laboratory environment it is `odom`.

After using the Add button to add the specific visualization, sometimes it necessary to configure it, usually by specifying which topic you want to visualize.

http://wiki.ros.org/rviz/UserGuide



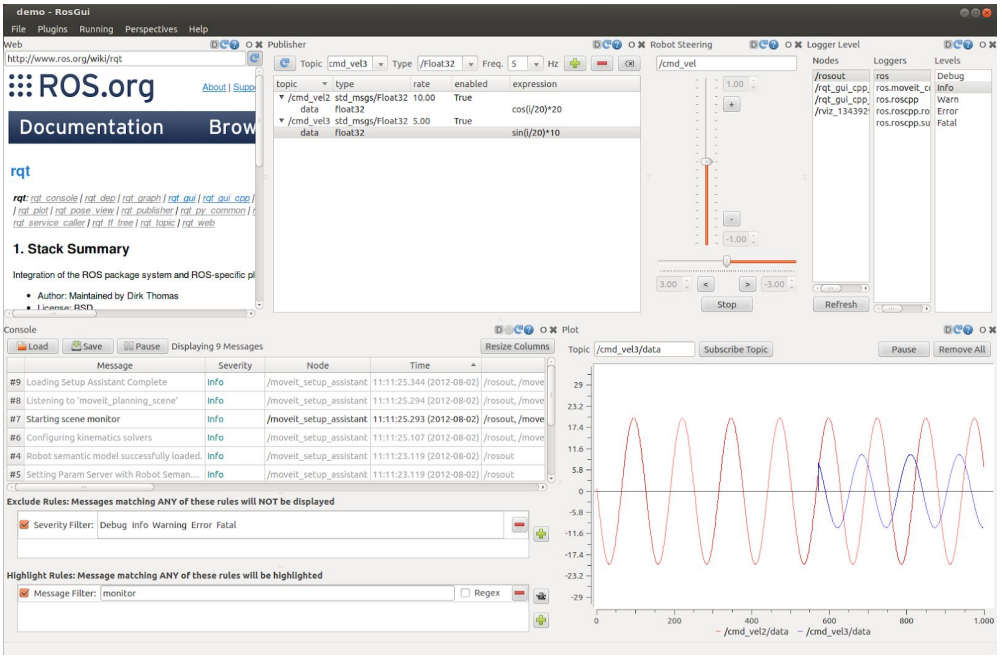
Commands:

```
rviz # Run Rviz
```

rqt

rqt is a software framework of ROS that implements the various GUI tools in the form of plugins. Example plugins: node graph, camera visualizer.

http://wiki.ros.org/rqt



rosshow

Using rosshow you can visualize LaserScans or other types of messages in the terminal.

https://github.com/dheera/rosshow

Commands:

```
rosrun rosshow rosshow <topicname> # Run Rosshow
```