Robotics: Report

a ROS-based prototype of a robotic arm for train inspection

Ciardi Roberto, Falzone Giovanni

jointly M.Sc Embedded Computing Systems

University of Pisa

Sant'Anna School of Advanced Studies

April 01, 2019

Contents

Abstract

This document describes the process of realization of a robotic arm prototype, designed using ROS[1]. At first an introduction of the problem of mechanical environment inspection is given, underlying which are the purposes of the designed system: a robotic arms that aims at moving below a train axis, to check pads status using sensors. After that, the architecture of the system is shown, describing further details about each component as the world creation, the arm design and its movements, and the communications mainly provided by ros topics. Lastly a brief guide on how to execute the system is shown, then conclusions remark the work made and the possible modifications and updates that could be done on it.

The appendices are also important to add details about *SDF*, *xacro* and *urdf* files, used to design each model and also to describe *MoveIt framework* and *PCL sample consensus*, widely used in ROS projects and very useful for our work.

The implementation of the project and the *robotic_ arm_ inspector* directory can be found as a git repository at `https://github.com/GiovanniFalzone/robotic_arm_inspector`.

[1]Robot Operative System: `http://www.ros.org/`

Introduction

The project represents a ROS prototype for a robotic arm, useful for works of inspection of mechanical environments, in this case with particular attention to train inspection.

Overview of the problem

The checking of trains' **brake pads** is not of easy accomplishment and could be dangerous for a man to move below a train for these kind of inspections: the purpose of the project is to create a robotic arm capable of moving below a train and automatically, or by remote control, finding and checking brake pads status, eliminating users' direct interaction with train mechanisms. Brake pads are naturally a critical aspect for trains as in general for vehicles, and this robotic arm has the duty to check the **width** of pads, to understand if they are in a good status or should be replaced.

Overview of the system

The system is mainly divided in two parts: the part of design and rendering of **Gazebo** world, with robotic arm and train wheel axis, and the part of world analysis and arm motion across the world, to perform the inspection. The project involves the use of two main models: a train model, with brake pads to be inspected, and a **UR5** robotic arm with a **kinect** mounted on its end effector. The user must interact with the system to create a train model and send instructions to the robotic arm to inspect brake pads.

Models generation

To design the system we firstly needed to design a suitable Gazebo world, with models usable on Gazebo and RVIZ. In particular we needed two models:

- **Train Model**, that is static and must be inspected by the robotic arm
- **Robotic arm Model**, that is dynamic, can be moved by instructions and, using kinect, can inspect the environment and the train model

These two models were created in a different way, due to their different uses: in particular the train model must only be rendered in **Gazebo** environment, whereas the robotic arm is a composition of models and should receive instructions from users.

Train Model

The model of the train was created to be inspected by the robotic arm. Only the wheel axis has been designed, avoiding the train chassis, unuseful for our purposes: measurements were chosen to have a model the more possible adherent to reality. A first skeleton was created as an **sdf**[2] file, whereas the execution of the program allows user to spawn trains with different characteristics with respect to pad status, to observe different behaviors of the robotic arm.

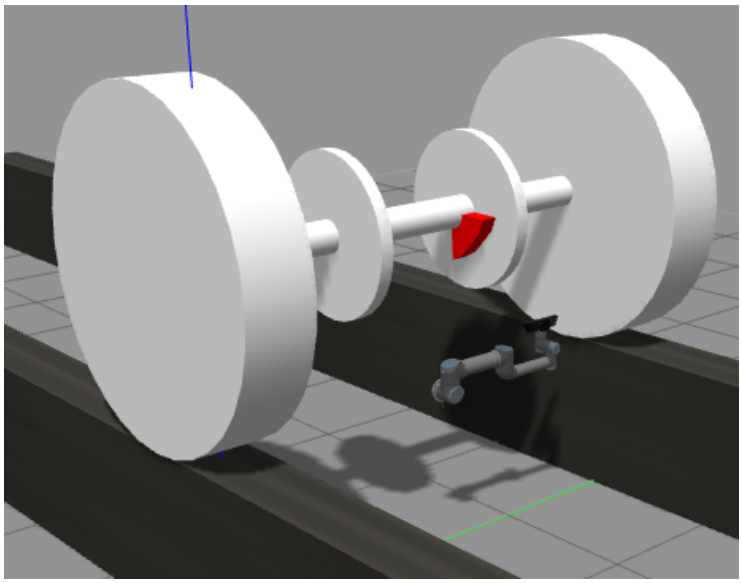


Figure 1: A train model with red pads

The train model was designed as a composition of links in a sdf file and it contains:

- Wheel axis, designed as a three meters cylinder[3], with the other parts mounted on it

[2]Extension for xml file to design model for gazebo: http://sdformat.org/spec

[3]A cylinder link has a pose tag that indicates the pose of its center, a radius and an height that represents its width. Further details are given in appendix A.

- Two wheels, designed as cylinders centered at the end of the axis
- Two disks, designed as cylinders centered at one third and two thirds of the axis
- Two pads, designed as polylines[4], attached to disks and with variable widths, choosable at runtime

The detailed process of creation of train model is given in the appendix (A), with further explanation about sdf models and their characteristics.

Robotic Arm Model

The model of the robot is described in a *.xacro.urdf* file in which three components can be distinguished:

- cart, that is just a link for future purpose
- UR5 model, Universal Robot 6-DOF arm model [4]
- Kinect sensor model [5]

The last two models are defined as *xacro.urdf* by the author of each package. The UR5 is composed by 6 rotating joints(limited to move within a $\pm\pi$ angle) and is able to move 5Kg of payload, and its characteristics are specified in the official .urdf model of the arm. The UR5 model is attached to the cart joint and then the kinect is attached to the *End Effector joint* of the UR5 realizing the final chain of the robot.

Executing the project the *.xacro.urdf*[5] is built in a *.urdf*[6] file that contains the description of the final robot as composition of each component. Further details and explanation about xacro file and compositions can be found in appendix B.

[4] A polyline link has a pose that indicates the position of its upper-left vertex, and is designed as a serie of points. Further details are given in appendix A.

[5] Xacro (XML Macros) Xacro is an XML macro language: http://wiki.ros.org/xacro

[6] Unified Robot Description Format: http://wiki.ros.org/urdf

Architecture

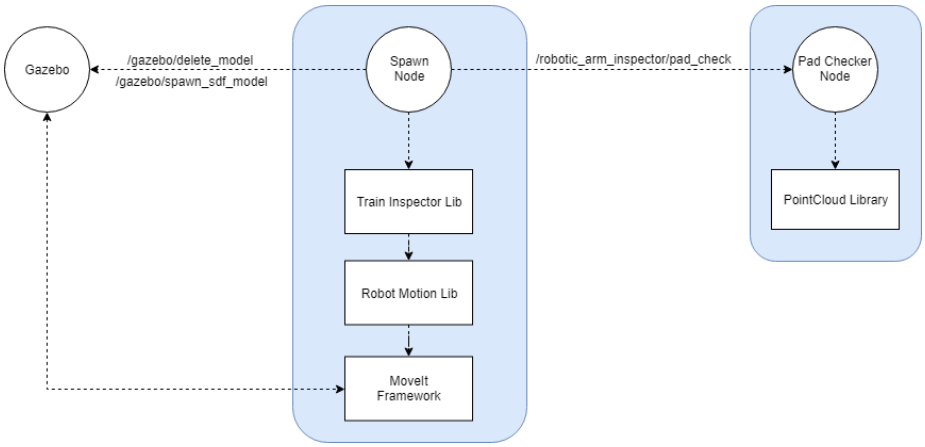


Figure 2: System Architecture

In order to divide the tasks in different nodes and to study and use the ROS components as Topics, Messages, Nodes etc., the Architecture of the system is composed by two main nodes as can be seen in the figure 2:

- **Spawn Train**, used to spawn train models in Gazebo world, with different features(different pads dimension) and successively start the software to inspect the environment and execute the movement algorithm, to accomplish the desired task.
- **Pad Checker**, subscribed to a topic in order to receive a **Point Cloud** related to a pad,and check pads' dimensions.

Spawn train

This node implements a CLI in python, to allow user to spawn a train model, with chosen characteristics, and move the arm to inspect it. As we can see

```
[INFO] [1554192399.874849, 16.834000]: Waiting for /gazebo/delete_model serv
..
[INFO] [1554192399.880927, 16.841000]: Connected to delete service!
[INFO] [1554192399.881307, 16.841000]: Waiting for /gazebo/spawn_sdf_model s
ce...
[INFO] [1554192399.883226, 16.843000]: Connected to spawn service!
Insert a command:
        -> "good" to spawn a train with "good" pads
        -> "bad" to spawn a train with "bad" pads
        -> "rand" to spawn a train with random pads
        -> "custom" to spawn a train with chosen pads width
        -> "continue" to continue (train already spawned)
good
Spawning train with good pads
Insert a command
        -> "del" to delete the spawned train
        -> "print" to print train specs
        -> "confirm" to confirm and continue
confirm
```

Figure 3: Spawn train cli

in figure 3 the user can spawn different kind of trains, according to their pads

status[7]. To render these different models we have used **rospy**, a ROS python client library, that allows to interact with gazebo services, spawning or deleting models directly on a gazebo world.

At first the sdf model of the train is parsed to create a python dictionary for the train, that contains train's characteristics, and that is eventually used to create the sdf for a new train with different characteristics. Starting from an sdf file (the one ready-to-use or the one created programmatically) the model can be spawn on Gazebo world, using **gazebo services**: ROS services[8] are defined by srv files which contains a request and a response message, identical to messages uses with ROS topics. A service can be called, creating a **rospy.ServiceProxy** with the name of the service needed. In particular in this case we've used spawn_sdf_model and delete_model services as we can see in *spawn_ train.py*:

```
delete_srv = rospy.ServiceProxy('/gazebo/delete_model', DeleteModel)
rospy.loginfo("Waiting for /gazebo/delete_model service...")
delete_srv.wait_for_service()
rospy.loginfo("Connected to delete service!")

spawn_srv = rospy.ServiceProxy('/gazebo/spawn_sdf_model', SpawnModel)
rospy.loginfo("Waiting for /gazebo/spawn_sdf_model service...")
spawn_srv.wait_for_service()
rospy.loginfo("Connected to spawn service!")
```

These two services are obviously used to delete and spawn models on gazebo.

Every time a train model is spawned it is used by the `train_inspector` class in order to know the feature of the train model as in the sdf file. This node implements the actions to move the robotic arm to inspect the train and the environment, those methods were implemented in the *train_ inspector* class and can be used through the *CLI*, as seen in 5. Some of them are:

- check environment, to check the environment and make the kinect build an octree
- inspect pad, to check pad width
- inspect axis, to move the arm along the axis
- set train model, to choose a new train model

When a train is spawned the first action done by the robot is to check the environment in order to create a basic knowledge of the train and the obstacles. The environment created by this inspection can be seen in figure 4 that represents the RVIZ motion planning. Then tha user can choose an action for the arm e.g. the pad check. The pad check move the robot close to the pad using the `robot_motion_lib` class, then collect the cloud point received via the topic `/camera1/depth/points` associated with the kinect sensor. The collected Cloud Point is sent as a custom message via the `/robotic_arm_inspector/pad_checker` topic, to be processed.

[7]We've chosen a maximum pads width of 0.1m and a minimum of 0.01m

[8]A list of services can be seen with the command **rosservice list** that returns the active services, with gazebo services that are listed as *gazebo/servicename*.

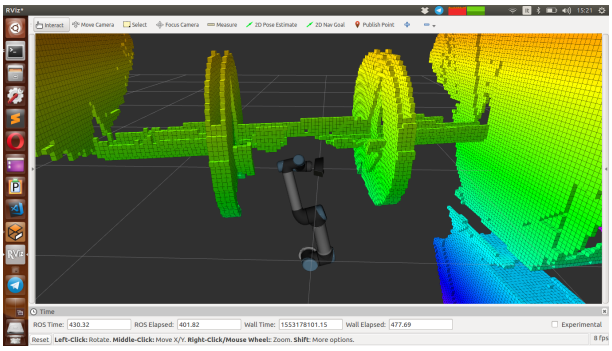


Figure 4: RVIZ motion planning

```
confirm
[INFO] [1554192407.306854, 23.804000]: Inspecting the environment
[INFO] [1554192441.042127, 23.806000]: Success
Insert a command:
        -> "init" to move to initial position
        -> "max" to move to max estension position
        -> "line" to move following a cone base
        -> "circle" to move following a circle
        -> "cone" to move in checking position
        -> "pad" to check pads
        -> "check" to check the environment
        -> "pos" to move the arm in a specific position
        -> "rotate" to rotate the end effector
        -> "print" to print train specs
        -> "restart" to delete the train and spawn a new train
```

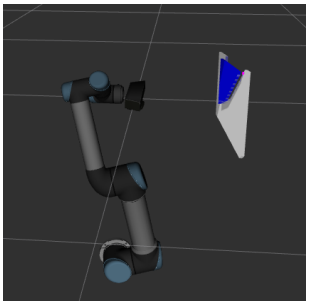
Figure 5: Train inspector CLI

Train inspector library

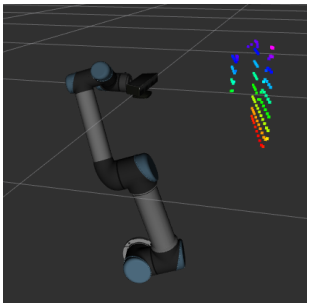
This library is used to realize the abstraction level for the inspector robot, it receives the robot model and implements some methods in order to move the arm below the train and inspect the reachable point. It has a member that is a `robot_motion_lib` class instance in order to apply some kind of motion as move following a line, a circle or a cone.

Robot motion library

This library is used to realize the abstaction level for the general motion and to implement some pre-defined movement. The functions have as arguments the xyz vector (position) and the rpy vector (rotation) used to plan and, if possible, execute a motion to reach that specific point. This library is realized on top of the **MoveIt API**: by defining a position goal or a joints goal the MoveIt framework tries to find a motion plan to move the robot in that goal position avoiding the known obstacles, as the robot itself and the environment represented as the Octree map shown before 4. The Kinematic of the robot and the obstacle avoidance is implemented in the MoveIt framework: further details are given in appendix C.



(a) Point cloud received



(b) Point cloud filtered

Pad checker

To realize this node we've chosen cpp instead of python in order to use the **PCL** (Point Cloud Library). This node aims at receiving a point cloud of the pad, from the `/robotic_arm_inspector/pad_checker` topic, and execute these actions:

- it **filters** the Point Cloud with a **passtrough filter**, to remove out of range points
- it **filter** the Point Cloud with a **Voxel filter**, to reduce the number of points (figure 6b)
- it finds the first largest plane in the filtered Cloud Point (figure 8a)
- it removes the inliers related to the first plane from the filtered Cloud Point
- it finds the second largest plane in the remaining points (figure 8b)
- it computes planes'distance, to find pad width. Computed as average between the distances between each inlier of the first plane and the second plane equation.

The result is given in command line as seen in figure 7. Further details about

```
rob@rob-Aspire-A315-53G:~$ rosrun robotic_arm_inspector pad_checker_node
[ INFO] [1554192508.536328808, 117.349000000]: Pad width: 0.012870
[ INFO] [1554192508.536434639, 117.349000000]: WARNING: pad should be replaced
```

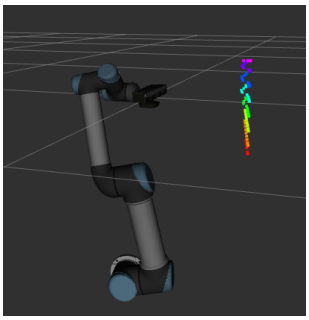
Figure 7: Pad checker result for a bad pad

PCL and Sample Consensus are given in appendix D

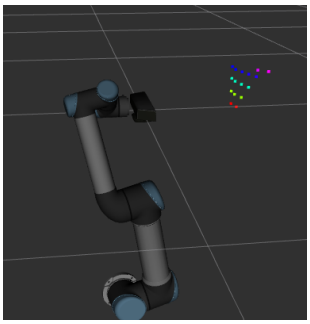
Architecture topics

Each component of the architecture, included the MoveIt components, communicate via topics and defined messages, thanks to that each component knows robot's status, sensors' data, motion planning status etc.

In the figure 9 a full vision of the topics used in our architecutre is reported: as we can see there are some components related to the MoveIt framework,



(a) First plane



(b) Second plane

to publish the status of the robot and receive the requests of movement and planning from the top part of the architecture, in this way is much easier to focus on the application part of the robot and leave to MoveIt the role of managing the kinematics of the robot.

Another important component is the **Camera1** topic that is related to the kinect, and is addressed to publish informations retrieved by the kinect's sensors. The graph shows just the `/camera1/depth/points` that is related to the Point Cloud, but there are much more topics (as rgb cam) that we are not using in this version of the applications, all of them are implemented by the **openni** plugin that simulates the kinect in the environment.

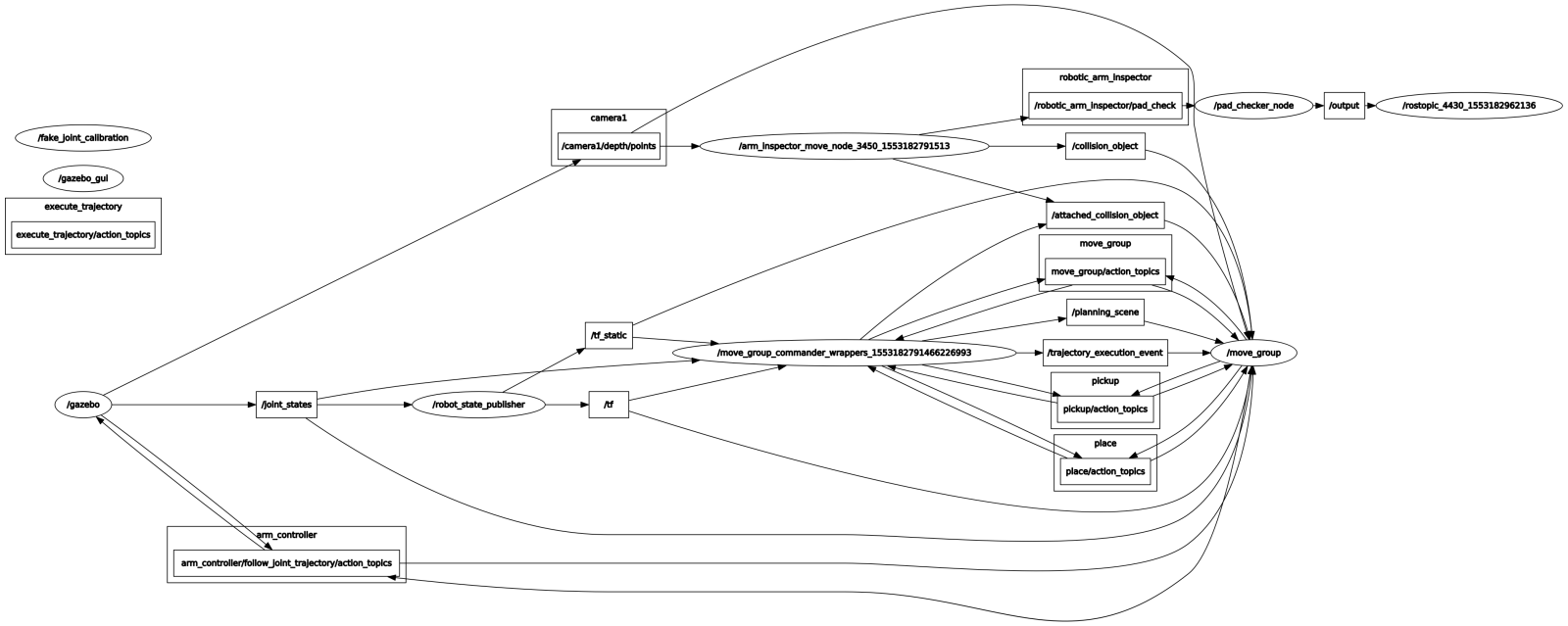


Figure 9: Topics of the system

How to execute the project

The project is contained in a ROS package named "robotic_arm_inspector" and it can be launched with the command:

```
$ roslaunch robotic_arm_inspector init_robotic_arm_inspector launch
```

This command will open Gazebo and railway.world. In three other terminals we must execute ROS nodes: rviz, spawn_train python script and pad_checker_node. So we must execute these commands:

```
$ rosrun rviz rviz
$ rosrun robotic_arm_inspector pad_checker_node
$ rosrun robotic_arm_inspector spawn_train py
```

Here we give a brief resume of the functionalities of each of these rosnodes:

- RVIZ allows to see the motion planning and the cloud points generated using the kinect, while moving the UR5 robotic arm
- pad_checker_node computes pad width, by using the information given by the point cloud generated with kinect
- spawn_train is responsible for user interaction and allows to spawn a train model and move the arm, using some line commands

Conclusions

Our goal was to face a real problem scenario and realize a simulation environment able to represent the problem, to build and test a possible solution. Using ROS together with a set of frameworks and packages was a keypoint of the project, to see how this kind of application are used in a real world scenario. Starting with industrial components as UR5 robotic and a kinect, that are widely used in robotics applications, we was able to build from skretch a possible solution to the initial problem and study how ROS is used in robotics application, to connect each needed component for this kind of applications.

Another important point was the usage of MoveIt framework to build an high level API useful to act on the real robot and essentially to integrate sensors data, to have a knowledge of the environment and apply some **collision avoidance** algorithms togheter with the motion plan algorithm.

Possible future works

This work was just an initial approach to this problem, but was essential for us to study the state of the art in Robotics applications. This *Beta* version of the *Robotic arm inspector* could be improved by applying some of these feature:

- change set the base joint of the arm as not fixed joint in order to be a more realistic world example
- add a base cart model to allow the movement of the robotic arm parallel to the railway
- allow the robotic arm base joint to move perpendicular to the rails in order to reach all the under train points
- implement our own Kinematic solver for the final robot including the modification in this list
- improve the pads models to fit better the real situation
- realize an AI based pad checker able to find the not absolute pad's positions and detect pads imperfections
- randomize the below train environment to insert random obstacles for cart and arm movements

Appendices

SDF file

Sdf files are **xml** file useful to render a model on gazebo. Gazebo measurements are in meters, so the choices made to implement the models we've used, were based on this assumption, trying to create realistic models that can help visualizing the problem in the best way as possible. In the subdirectory of the project *models* we can find a directory for tracks model and one for train model. A mention for **tracks model** is needed: it represents the tracks as two simple black parallelepipeds(figure 10) where the train will be layed on and with the robotic arm that will lay between them as we can see in figure 1) The train

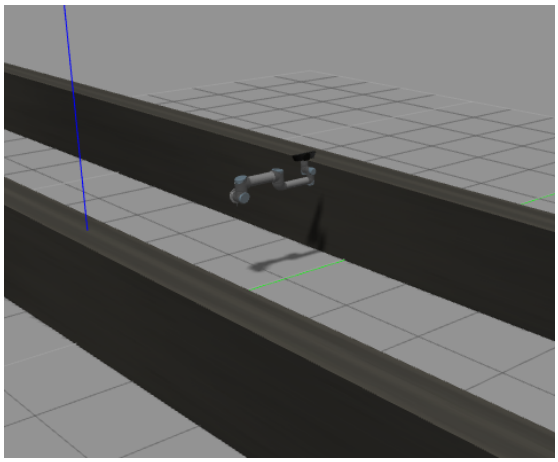


Figure 10: Tracks model as shown in "railway.world"

model was firstly created as an **rsdf** file that allows to use variables, useful to calculate distances between components on the wheel axis. In particular the train has an axis, that is a thin cylinder, two wheels at the begin and the end of the axis, two disks that are set respectively at one third and two thirds of the axis and two pads that, as default, have a width of 0.1 m and are shaped as real brake pads, using polyline objects of sdf and laying near to the disks. In this portion of *models/train_ model/model.rsdf* file we can see how position arrays for wheels and other cylinders are created (a position array contains x,y,z coordinates and roll, pitch, yaw rotations):

```
    x0 = 0.0
    z0 = wheel_radius + bin_height
    y0 = 0.0
    roll = 0.0
    pitch = Math::PI/2
    yaw = Math::PI/2

    #wheel variables
    lw_y0 = 0.0
    left_wheel_arr = [x0, lw_y0, z0, roll, pitch, yaw]

    rw_y0 = axis_width
```

```
    right_wheel_arr = [x0, rw_y0, z0,roll, pitch, yaw]

    #axis variables
    ax_y0 = (rw_y0 - lw_y0)/2
    axis_arr = [x0, ax_y0, z0, roll, pitch, yaw]

    #disk variables
    ld_y0 = axis_width/3
    left_disk_arr = [x0, ld_y0, z0, roll, pitch, yaw]

    rd_y0 = axis_width*2/3
    right_disk_arr = [x0, rd_y0, z0, roll, pitch, yaw]
```

The above arrays are then used in the xml part of the file to specify the <**pose**> tag, that contains the coordinates of one of the point of the figure: for a cylinder it indicates the center of the cylinder, for a polyline it indicates the upper-left point. In this part of *model.rsdf* we can see the creation of the link for left_wheel with a collision object that is basically the same as the visual object and with characteristics that are specified as variables, as seen before:

```
  <link name="left_wheel">
  <pose><%= left_wheel_arr.join(" ") %></pose>
  <collision name="collision">
    <geometry>
      <cylinder>
        <radius><%= wheel_radius %></radius>
        <length><%= wheel_width %></length>
      </cylinder>
    </geometry>
  </collision>
  <visual name="visual">
    <geometry>
      <cylinder>
        <radius><%= wheel_radius %></radius>
        <length><%= wheel_width %></length>
      </cylinder>
    </geometry>
  </visual>
</link>
```

Through command line, the sdf file is then created:

```
$ erb model.rsdf > model.sdf
```

Once the model.sdf file is created, it can be copied in gazebo models directory[9], that contains gazebo models that can be rendered directly on gazebo, from the model menu. In our case the model will be automatically spawned, not requesting a direct interaction with gazebo.

[9] */home/user/.gazebo/models*

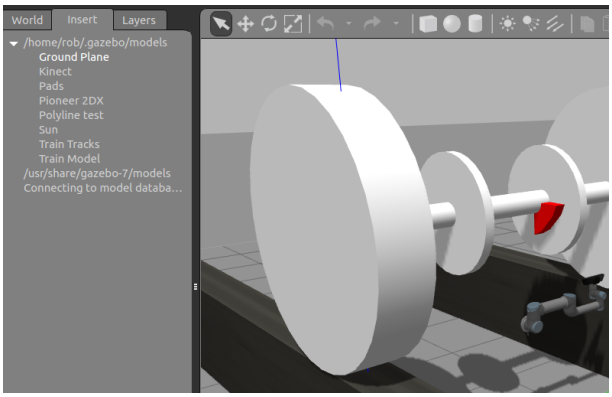


Figure 11: Train Model manually inserted from "insert" menu

Xacro model

Robotic Arm Inspector model composition

As said the final model of the Robotic Arm Inspector is described using a *.xacro.urdf* file, in which the robot is described as a composition of different models in a fashion, scalable and hierachical way. In the following piece of code, the first part of the *.xacro.urdf* file is reported: here the arm inspector is defined and we can distinguish the cart and the two used models.

```
<?xml version="1.0"?>
<robot name="robotic_arm_inspector" xmlns:xacro="http://ros.org/wiki/xacro"
    >
<!-- ur5 from universal robots package-->
<xacro:include filename="$(find ur_description)/urdf/common.gazebo.xacro" /
    >
<xacro:include filename="$(find ur_description)/urdf/ur5.urdf.xacro" />
<!-- kinect -->
<xacro:include filename="$(find common_sensors)/urdf/sensors/kinect.urdf.
    xacro" />
<!-- <xacro:include filename="$(find common_sensors)/urdf/sensors/kinect2.
    urdf.xacro" /> -->

<link name="world" />

<joint name="world_joint" type="fixed">
  <parent link="world" />
  <child link = "cart" />
  <origin xyz="0.0 0.0 1.0"/>
</joint>

<link name = "cart"/>

<joint name="cart_joint" type="fixed">
  <parent link="cart" />
  <child link = "base_link" />
  <origin xyz="0.0 1.5 0.0" rpy="1.570796 0.0 1.570796"/>
</joint>

<xacro:ur5_robot prefix="" joint_limited="true"
  shoulder_pan_lower_limit="${-pi}" shoulder_pan_upper_limit="${pi}"
  shoulder_lift_lower_limit="${-pi}" shoulder_lift_upper_limit="${pi}"
  elbow_joint_lower_limit="${-pi}" elbow_joint_upper_limit="${pi}"
  wrist_1_lower_limit="${-pi}" wrist_1_upper_limit="${pi}"
  wrist_2_lower_limit="${-pi}" wrist_2_upper_limit="${pi}"
  wrist_3_lower_limit="${-pi}" wrist_3_upper_limit="${pi}"
/>
<!-- Attach Kinect -->
<xacro:sensor_kinect parent="ee_link" />
<!-- <xacro:kinect2 parent="ee_link" name="KinectSensor2">
  <origin xyz="0 0 0" rpy="0 0 0" />
</xacro:kinect2> -->

</robot>
```

UR5 model

Here a part of the original *.xacro.urdf* description of the robotic arm is reported: initially the phisical features of the robot, used by the other tools to simulate

and move it correctly, are described. The entire file is attached in subsection B.4, where we can see how the robotic arm model, with its parts as joints and links, is described in a fashion, hierarchical way using the xml language.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://wiki.ros.org/xacro">

  <xacro:include filename="$(find ur_description)/urdf/ur.transmission.xacro
      " />
  <xacro:include filename="$(find ur_description)/urdf/ur.gazebo.xacro" />

  <xacro:macro name="cylinder_inertial" params="radius length mass *origin">
  <inertial>
    <mass value="${mass}" />
    <xacro:insert_block name="origin" />
    <inertia ixx="${0.0833333 * mass * (3 * radius * radius + length *
        length)}" ixy="0.0" ixz="0.0"
    iyy="${0.0833333 * mass * (3 * radius * radius + length * length)}" iyz=
        "0.0"
    izz="${0.5 * mass * radius * radius}" />
  </inertial>
  </xacro:macro>

  <xacro:macro name="ur5_robot" params="prefix joint_limited
   shoulder_pan_lower_limit:=${-pi} shoulder_pan_upper_limit:=${pi}
   shoulder_lift_lower_limit:=${-pi} shoulder_lift_upper_limit:=${pi}
   elbow_joint_lower_limit:=${-pi} elbow_joint_upper_limit:=${pi}
   wrist_1_lower_limit:=${-pi} wrist_1_upper_limit:=${pi}
   wrist_2_lower_limit:=${-pi} wrist_2_upper_limit:=${pi}
   wrist_3_lower_limit:=${-pi} wrist_3_upper_limit:=${pi}">

  <!-- Inertia parameters -->
  <xacro:property name="base_mass" value="4.0" /> <!-- This mass might be
      incorrect -->
  <xacro:property name="shoulder_mass" value="3.7000" />
  <xacro:property name="upper_arm_mass" value="8.3930" />
  <xacro:property name="forearm_mass" value="2.2750" />
  <xacro:property name="wrist_1_mass" value="1.2190" />
  <xacro:property name="wrist_2_mass" value="1.2190" />
  <xacro:property name="wrist_3_mass" value="0.1879" />

  <xacro:property name="shoulder_cog" value="0.0 0.00193 -0.02561" />
  <xacro:property name="upper_arm_cog" value="0.0 -0.024201 0.2125" />
  <xacro:property name="forearm_cog" value="0.0 0.0265 0.11993" />
  <xacro:property name="wrist_1_cog" value="0.0 0.110949 0.01634" />
  <xacro:property name="wrist_2_cog" value="0.0 0.0018 0.11099" />
  <xacro:property name="wrist_3_cog" value="0.0 0.001159 0.0" />

  <!-- Kinematic model -->
  <!-- Properties from urcontrol.conf -->
  <!--
    DH for UR5:
    a = [0.00000, -0.42500, -0.39225, 0.00000, 0.00000, 0.0000]
    d = [0.089159, 0.00000, 0.00000, 0.10915, 0.09465, 0.0823]
    alpha = [ 1.570796327, 0, 0, 1.570796327, -1.570796327, 0 ]
    q_home_offset = [0, -1.570796327, 0, -1.570796327, 0, 0]
    joint_direction = [-1, -1, 1, 1, 1, 1]
    mass = [3.7000, 8.3930, 2.2750, 1.2190, 1.2190, 0.1879]
    center_of_mass = [ [0, -0.02561, 0.00193], [0.2125, 0, 0.11336],
        [0.11993, 0.0, 0.0265], [0, -0.0018, 0.01634], [0, 0.0018,0.01634],
         [0, 0, -0.001159] ]
  -->
```

```
<xacro:property name="d1" value="0.089159" />
<xacro:property name="a2" value="-0.42500" />
<xacro:property name="a3" value="-0.39225" />
<xacro:property name="d4" value="0.10915" />
<xacro:property name="d5" value="0.09465" />
<xacro:property name="d6" value="0.0823" />

<!-- Arbitrary offsets for shoulder/elbow joints -->
<xacro:property name="shoulder_offset" value="0.13585" /> <!-- measured
    from model -->
<xacro:property name="elbow_offset" value="-0.1197" /> <!-- measured from
    model -->

<!-- link lengths used in model -->
<xacro:property name="shoulder_height" value="${d1}" />
<xacro:property name="upper_arm_length" value="${-a2}" />
<xacro:property name="forearm_length" value="${-a3}" />
<xacro:property name="wrist_1_length" value="${d4 - elbow_offset -
    shoulder_offset}" />
<xacro:property name="wrist_2_length" value="${d5}" />
<xacro:property name="wrist_3_length" value="${d6}" />
<!--property name="shoulder_height" value="0.089159" /-->
<!--property name="shoulder_offset" value="0.13585" /--> <!--
    shoulder_offset - elbow_offset + wrist_1_length = 0.10915 -->
<!--property name="upper_arm_length" value="0.42500" /-->
<!--property name="elbow_offset" value="0.1197" /--> <!-- CAD measured -->
<!--property name="forearm_length" value="0.39225" /-->
<!--property name="wrist_1_length" value="0.093" /--> <!-- CAD measured --
    >
<!--property name="wrist_2_length" value="0.09465" /--> <!-- In CAD this
    distance is 0.930, but in the spec it is 0.09465 -->
<!--property name="wrist_3_length" value="0.0823" /-->

<xacro:property name="shoulder_radius" value="0.060" /> <!-- manually
    measured -->
<xacro:property name="upper_arm_radius" value="0.054" /> <!-- manually
    measured -->
<xacro:property name="elbow_radius" value="0.060" /> <!-- manually
    measured -->
<xacro:property name="forearm_radius" value="0.040" /> <!-- manually
    measured -->
<xacro:property name="wrist_radius" value="0.045" /> <!-- manually
    measured -->
[...]
```

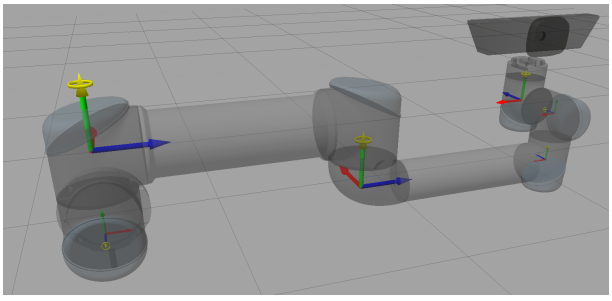


Figure 12: Joints view of the UR5 robotic arm

In figure 12 the position of each joint's frame is shown, whereas in figure 12 is shown the center of mass of each component. As said each component is

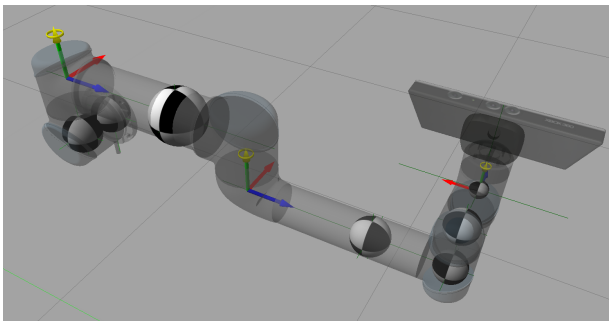


Figure 13: Center of mass the UR5 robotic arm's components

described in the *.xacro* file in a hierarchical way and all of this characteristics are described there.

Kinect sensor model

In the same way the kinect is described as phisical model in order to simulate the real scenario in which the kinect is a phisical component and can seen as a payload attached to the robotic arm. The kinect is also

```
<?xml version="1.0"?>
<root name="sensor_kinect" xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="$(find common_sensors)/urdf/sensors/
       kinect_gazebo.xacro" />
  <xacro:include filename="$(find common_sensors)/urdf/sensors/
       kinect_properties.urdf.xacro"/>

  <!-- camera_name has to be unique! -->
  <xacro:kinect_sensor link_name="camera_depth_frame" camera_name="camera1
       "
            frame_name="camera_depth_optical_frame"/>

  <!-- Parameterised in part by the values in kinect_properties.urdf.xacro
        -->
  <xacro:macro name="sensor_kinect" params="parent">
    <joint name="camera_rgb_joint" type="fixed">
        <origin xyz="${cam_px} ${cam_py} ${cam_pz}"
          rpy="${cam_or} ${cam_op} ${cam_oy}"/>
      <parent link="${parent}"/>
      <child link="camera_rgb_frame" />
    </joint>
    <link name="camera_rgb_frame">
      <inertial>
        <mass value="0.001" />
        <origin xyz="0 0 0" />
        <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0"
            izz="0.0001" />
      </inertial>
    </link>
    <joint name="camera_rgb_optical_joint" type="fixed">
      <origin xyz="0 0 0" rpy="${-M_PI/2} 0 ${-M_PI/2}" />
      <parent link="camera_rgb_frame" />
      <child link="camera_rgb_optical_frame" />
    </joint>
    <link name="camera_rgb_optical_frame">
```

```
    <inertial>
      <mass value="0.001" />
      <origin xyz="0 0 0" />
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0"
          izz="0.0001" />
    </inertial>
  </link>

  <joint name="camera_joint" type="fixed">
    <origin xyz="-0.031 ${-cam_py} -0.016" rpy="0 0 0"/>
    <parent link="camera_rgb_frame"/>
    <child link="camera_link"/>
  </joint>

  <link name="camera_link">
    <visual>
     <origin xyz="0 0 0" rpy="0 0 ${M_PI/2}"/>
      <geometry>
      <mesh filename="package://common_sensors/meshes/sensors/kinect.dae
          "/>
      </geometry>
    </visual>
    <collision>
      <origin xyz="0.0 0.0 0.0" rpy="0 0 0"/>
      <geometry>
        <box size="0.07271 0.27794 0.073"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.001" />
      <origin xyz="0 0 0" />
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0"
          izz="0.0001" />
    </inertial>
  </link>

  <!-- The fixed joints & links below are usually published by
       static_transformers launched by the OpenNi launch
   files. However, for Gazebo simulation we need them, so we add them
       here.
   (Hence, don't publish them additionally!) -->
  <joint name="camera_depth_joint" type="fixed">
    <origin xyz="0 ${2 * -cam_py} 0" rpy="0 0 0" />
    <parent link="camera_rgb_frame" />
    <child link="camera_depth_frame" />
  </joint>
  <link name="camera_depth_frame">
    <inertial>
      <mass value="0.001" />
      <origin xyz="0 0 0" />
      <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0"
          izz="0.0001" />
    </inertial>
  </link>
  <joint name="camera_depth_optical_joint" type="fixed">
    <origin xyz="0 0 0" rpy="${-M_PI/2} 0 ${-M_PI/2}" />
    <parent link="camera_depth_frame" />
    <child link="camera_depth_optical_frame" />
  </joint>
  <link name="camera_depth_optical_frame">
    <inertial>
      <mass value="0.001" />
```

```
          <origin xyz="0 0 0" />
          <inertia ixx="0.0001" ixy="0.0" ixz="0.0" iyy="0.0001" iyz="0.0"
               izz="0.0001" />
        </inertial>
      </link>
    </xacro:macro>
  </root>
```

UR5 Xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://wiki.ros.org/xacro">

<xacro:include filename="$(find ur_description)/urdf/ur.transmission.xacro"
      />
<xacro:include filename="$(find ur_description)/urdf/ur.gazebo.xacro" />

<xacro:macro name="cylinder_inertial" params="radius length mass *origin">
  <inertial>
  <mass value="${mass}" />
  <xacro:insert_block name="origin" />
  <inertia ixx="${0.0833333 * mass * (3 * radius * radius + length * length)
      }" ixy="0.0" ixz="0.0"
    iyy="${0.0833333 * mass * (3 * radius * radius + length * length)}" iyz=
        "0.0"
    izz="${0.5 * mass * radius * radius}" />
  </inertial>
</xacro:macro>

<xacro:macro name="ur5_robot" params="prefix joint_limited
  shoulder_pan_lower_limit:=${-pi} shoulder_pan_upper_limit:=${pi}
  shoulder_lift_lower_limit:=${-pi} shoulder_lift_upper_limit:=${pi}
  elbow_joint_lower_limit:=${-pi} elbow_joint_upper_limit:=${pi}
  wrist_1_lower_limit:=${-pi} wrist_1_upper_limit:=${pi}
  wrist_2_lower_limit:=${-pi} wrist_2_upper_limit:=${pi}
  wrist_3_lower_limit:=${-pi} wrist_3_upper_limit:=${pi}">

  <!-- Inertia parameters -->
  <xacro:property name="base_mass" value="4.0" /> <!-- This mass might be
       incorrect -->
  <xacro:property name="shoulder_mass" value="3.7000" />
  <xacro:property name="upper_arm_mass" value="8.3930" />
  <xacro:property name="forearm_mass" value="2.2750" />
  <xacro:property name="wrist_1_mass" value="1.2190" />
  <xacro:property name="wrist_2_mass" value="1.2190" />
  <xacro:property name="wrist_3_mass" value="0.1879" />

  <xacro:property name="shoulder_cog" value="0.0 0.00193 -0.02561" />
  <xacro:property name="upper_arm_cog" value="0.0 -0.024201 0.2125" />
  <xacro:property name="forearm_cog" value="0.0 0.0265 0.11993" />
  <xacro:property name="wrist_1_cog" value="0.0 0.110949 0.01634" />
  <xacro:property name="wrist_2_cog" value="0.0 0.0018 0.11099" />
  <xacro:property name="wrist_3_cog" value="0.0 0.001159 0.0" />

  <!-- Kinematic model -->
  <!-- Properties from urcontrol.conf -->
  <!--
  DH for UR5:
  a = [0.00000, -0.42500, -0.39225, 0.00000, 0.00000, 0.0000]
  d = [0.089159, 0.00000, 0.00000, 0.10915, 0.09465, 0.0823]
  alpha = [ 1.570796327, 0, 0, 1.570796327, -1.570796327, 0 ]
```

```
    q_home_offset = [0, -1.570796327, 0, -1.570796327, 0, 0]
    joint_direction = [-1, -1, 1, 1, 1, 1]
    mass = [3.7000, 8.3930, 2.2750, 1.2190, 1.2190, 0.1879]
    center_of_mass = [ [0, -0.02561, 0.00193], [0.2125, 0, 0.11336], [0.11993,
         0.0, 0.0265], [0, -0.0018, 0.01634], [0, 0.0018,0.01634], [0, 0,
        -0.001159] ]
    -->
    <xacro:property name="d1" value="0.089159" />
    <xacro:property name="a2" value="-0.42500" />
    <xacro:property name="a3" value="-0.39225" />
    <xacro:property name="d4" value="0.10915" />
    <xacro:property name="d5" value="0.09465" />
    <xacro:property name="d6" value="0.0823" />

    <!-- Arbitrary offsets for shoulder/elbow joints -->
    <xacro:property name="shoulder_offset" value="0.13585" /> <!-- measured
        from model -->
    <xacro:property name="elbow_offset" value="-0.1197" /> <!-- measured from
        model -->

    <!-- link lengths used in model -->
    <xacro:property name="shoulder_height" value="${d1}" />
    <xacro:property name="upper_arm_length" value="${-a2}" />
    <xacro:property name="forearm_length" value="${-a3}" />
    <xacro:property name="wrist_1_length" value="${d4 - elbow_offset -
        shoulder_offset}" />
    <xacro:property name="wrist_2_length" value="${d5}" />
    <xacro:property name="wrist_3_length" value="${d6}" />
    <!--property name="shoulder_height" value="0.089159" /-->
    <!--property name="shoulder_offset" value="0.13585" /--> <!--
        shoulder_offset - elbow_offset + wrist_1_length = 0.10915 -->
    <!--property name="upper_arm_length" value="0.42500" /-->
    <!--property name="elbow_offset" value="0.1197" /--> <!-- CAD measured -->
    <!--property name="forearm_length" value="0.39225" /-->
    <!--property name="wrist_1_length" value="0.093" /--> <!-- CAD measured --
        >
    <!--property name="wrist_2_length" value="0.09465" /--> <!-- In CAD this
        distance is 0.930, but in the spec it is 0.09465 -->
    <!--property name="wrist_3_length" value="0.0823" /-->

    <xacro:property name="shoulder_radius" value="0.060" /> <!-- manually
        measured -->
    <xacro:property name="upper_arm_radius" value="0.054" /> <!-- manually
        measured -->
    <xacro:property name="elbow_radius" value="0.060" /> <!-- manually
        measured -->
    <xacro:property name="forearm_radius" value="0.040" /> <!-- manually
        measured -->
    <xacro:property name="wrist_radius" value="0.045" /> <!-- manually
        measured -->


    <link name="${prefix}base_link" >
    <visual>
      <geometry>
      <mesh filename="package://ur_description/meshes/ur5/visual/base.dae" />
      </geometry>
      <material name="LightGrey">
      <color rgba="0.7 0.7 0.7 1.0"/>
      </material>
    </visual>
    <collision>
```

```
  <geometry>
  <mesh filename="package://ur_description/meshes/ur5/collision/base.stl"
       />
  </geometry>
</collision>
<xacro:cylinder_inertial radius="0.06" length="0.05" mass="${base_mass}">
  <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
</xacro:cylinder_inertial>
</link>

<joint name="${prefix}shoulder_pan_joint" type="revolute">
<parent link="${prefix}base_link" />
<child link = "${prefix}shoulder_link" />
<origin xyz="0.0 0.0 ${shoulder_height}" rpy="0.0 0.0 0.0" />
<axis xyz="0 0 1" />
<xacro:unless value="${joint_limited}">
  <limit lower="${-2.0 * pi}" upper="${2.0 * pi}" effort="150.0" velocity=
       "3.15"/>
</xacro:unless>
<xacro:if value="${joint_limited}">
  <limit lower="${shoulder_pan_lower_limit}" upper="${
       shoulder_pan_upper_limit}" effort="150.0" velocity="3.15"/>
</xacro:if>
<dynamics damping="0.0" friction="0.0"/>
</joint>

<link name="${prefix}shoulder_link">
<visual>
  <geometry>
  <mesh filename="package://ur_description/meshes/ur5/visual/shoulder.dae"
       />
  </geometry>
  <material name="LightGrey">
  <color rgba="0.7 0.7 0.7 1.0"/>
  </material>
</visual>
<collision>
  <geometry>
  <mesh filename="package://ur_description/meshes/ur5/collision/shoulder.
       stl" />
  </geometry>
</collision>
<xacro:cylinder_inertial radius="0.06" length="0.15" mass="${shoulder_mass
    }">
  <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
</xacro:cylinder_inertial>
</link>

<joint name="${prefix}shoulder_lift_joint" type="revolute">
<parent link="${prefix}shoulder_link" />
<child link = "${prefix}upper_arm_link" />
<origin xyz="0.0 ${shoulder_offset} 0.0" rpy="0.0 ${pi / 2.0} 0.0" />
<axis xyz="0 1 0" />
<xacro:unless value="${joint_limited}">
  <limit lower="${-2.0 * pi}" upper="${2.0 * pi}" effort="150.0" velocity=
       "3.15"/>
</xacro:unless>
<xacro:if value="${joint_limited}">
  <limit lower="${shoulder_lift_lower_limit}" upper="${
       shoulder_lift_upper_limit}" effort="150.0" velocity="3.15"/>
</xacro:if>
<dynamics damping="0.0" friction="0.0"/>
```

```
  </joint>

  <link name="${prefix}upper_arm_link">
  <visual>
    <geometry>
    <mesh filename="package://ur_description/meshes/ur5/visual/upperarm.dae"
         />
    </geometry>
    <material name="LightGrey">
    <color rgba="0.7 0.7 0.7 1.0"/>
    </material>
  </visual>
  <collision>
    <geometry>
    <mesh filename="package://ur_description/meshes/ur5/collision/upperarm.
        stl" />
    </geometry>
  </collision>
  <xacro:cylinder_inertial radius="0.06" length="0.56" mass="${
      upper_arm_mass}">
    <origin xyz="0.0 0.0 0.28" rpy="0 0 0" />
  </xacro:cylinder_inertial>
  </link>

  <joint name="${prefix}elbow_joint" type="revolute">
  <parent link="${prefix}upper_arm_link" />
  <child link = "${prefix}forearm_link" />
  <origin xyz="0.0 ${elbow_offset} ${upper_arm_length}" rpy="0.0 0.0 0.0" />
  <axis xyz="0 1 0" />
  <xacro:unless value="${joint_limited}">
    <limit lower="${-pi}" upper="${pi}" effort="150.0" velocity="3.15"/>
  </xacro:unless>
  <xacro:if value="${joint_limited}">
    <limit lower="${elbow_joint_lower_limit}" upper="${
        elbow_joint_upper_limit}" effort="150.0" velocity="3.15"/>
  </xacro:if>
  <dynamics damping="0.0" friction="0.0"/>
  </joint>

  <link name="${prefix}forearm_link">
  <visual>
    <geometry>
    <mesh filename="package://ur_description/meshes/ur5/visual/forearm.dae"
         />
    </geometry>
    <material name="LightGrey">
    <color rgba="0.7 0.7 0.7 1.0"/>
    </material>
  </visual>
  <collision>
    <geometry>
    <mesh filename="package://ur_description/meshes/ur5/collision/forearm.
        stl" />
    </geometry>
  </collision>
  <xacro:cylinder_inertial radius="0.06" length="0.5" mass="${forearm_mass}"
      >
    <origin xyz="0.0 0.0 0.25" rpy="0 0 0" />
  </xacro:cylinder_inertial>
  </link>

  <joint name="${prefix}wrist_1_joint" type="revolute">
```

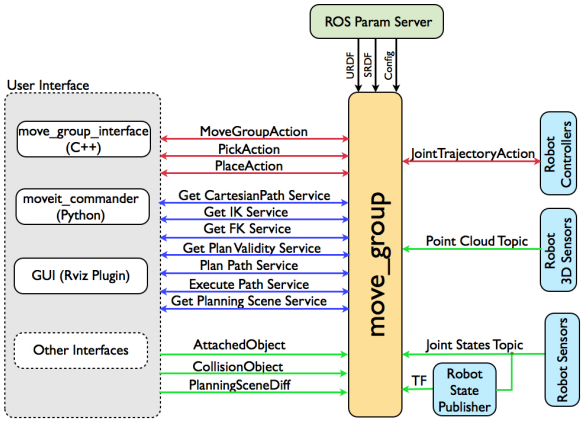
```
    <parent link="${prefix}forearm_link" />
    <child link = "${prefix}wrist_1_link" />
    <origin xyz="0.0 0.0 ${forearm_length}" rpy="0.0 ${pi / 2.0} 0.0" />
    <axis xyz="0 1 0" />
    <xacro:unless value="${joint_limited}">
      <limit lower="${-2.0 * pi}" upper="${2.0 * pi}" effort="28.0" velocity="
          3.2"/>
    </xacro:unless>
    <xacro:if value="${joint_limited}">
      <limit lower="${wrist_1_lower_limit}" upper="${wrist_1_upper_limit}"
          effort="28.0" velocity="3.2"/>
    </xacro:if>
    <dynamics damping="0.0" friction="0.0"/>
    </joint>

    <link name="${prefix}wrist_1_link">
    <visual>
      <geometry>
      <mesh filename="package://ur_description/meshes/ur5/visual/wrist1.dae" /
          >
      </geometry>
      <material name="LightGrey">
      <color rgba="0.7 0.7 0.7 1.0"/>
      </material>
    </visual>
    <collision>
      <geometry>
      <mesh filename="package://ur_description/meshes/ur5/collision/wrist1.stl
          " />
      </geometry>
    </collision>
    <xacro:cylinder_inertial radius="0.6" length="0.12" mass="${wrist_1_mass}"
        >
      <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
    </xacro:cylinder_inertial>
    </link>

    <joint name="${prefix}wrist_2_joint" type="revolute">
    <parent link="${prefix}wrist_1_link" />
    <child link = "${prefix}wrist_2_link" />
    <origin xyz="0.0 ${wrist_1_length} 0.0" rpy="0.0 0.0 0.0" />
    <axis xyz="0 0 1" />
    <xacro:unless value="${joint_limited}">
      <limit lower="${-2.0 * pi}" upper="${2.0 * pi}" effort="28.0" velocity="
          3.2"/>
    </xacro:unless>
    <xacro:if value="${joint_limited}">
      <limit lower="${wrist_2_lower_limit}" upper="${wrist_2_upper_limit}"
          effort="28.0" velocity="3.2"/>
    </xacro:if>
    <dynamics damping="0.0" friction="0.0"/>
    </joint>

    <link name="${prefix}wrist_2_link">
    <visual>
      <geometry>
      <mesh filename="package://ur_description/meshes/ur5/visual/wrist2.dae" /
          >
      </geometry>
      <material name="LightGrey">
      <color rgba="0.7 0.7 0.7 1.0"/>
      </material>
```

```
    </visual>
    <collision>
      <geometry>
      <mesh filename="package://ur_description/meshes/ur5/collision/wrist2.stl
           " />
      </geometry>
    </collision>
    <xacro:cylinder_inertial radius="0.6" length="0.12" mass="${wrist_2_mass}"
         >
      <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
    </xacro:cylinder_inertial>
    </link>

    <joint name="${prefix}wrist_3_joint" type="revolute">
    <parent link="${prefix}wrist_2_link" />
    <child link = "${prefix}wrist_3_link" />
    <origin xyz="0.0 0.0 ${wrist_2_length}" rpy="0.0 0.0 0.0" />
    <axis xyz="0 1 0" />
    <xacro:unless value="${joint_limited}">
      <limit lower="${-2.0 * pi}" upper="${2.0 * pi}" effort="28.0" velocity="
           3.2"/>
    </xacro:unless>
    <xacro:if value="${joint_limited}">
      <limit lower="${wrist_3_lower_limit}" upper="${wrist_3_upper_limit}"
           effort="28.0" velocity="3.2"/>
    </xacro:if>
    <dynamics damping="0.0" friction="0.0"/>
    </joint>

    <link name="${prefix}wrist_3_link">
    <visual>
      <geometry>
      <mesh filename="package://ur_description/meshes/ur5/visual/wrist3.dae" /
           >
      </geometry>
      <material name="LightGrey">
      <color rgba="0.7 0.7 0.7 1.0"/>
      </material>
    </visual>
    <collision>
      <geometry>
      <mesh filename="package://ur_description/meshes/ur5/collision/wrist3.stl
           " />
      </geometry>
    </collision>
    <xacro:cylinder_inertial radius="0.6" length="0.12" mass="${wrist_3_mass}"
         >
      <origin xyz="0.0 0.0 0.0" rpy="0 0 0" />
    </xacro:cylinder_inertial>
    </link>

    <joint name="${prefix}ee_fixed_joint" type="fixed">
    <parent link="${prefix}wrist_3_link" />
    <child link = "${prefix}ee_link" />
    <origin xyz="0.0 ${wrist_3_length} 0.0" rpy="0.0 0.0 ${pi/2.0}" />
    </joint>

    <link name="${prefix}ee_link">
    <collision>
      <geometry>
      <box size="0.01 0.01 0.01"/>
      </geometry>
```

```
      <origin rpy="0 0 0" xyz="-0.01 0 0"/>
    </collision>
    </link>

    <xacro:ur_arm_transmission prefix="${prefix}" />
    <xacro:ur_arm_gazebo prefix="${prefix}" />

    <!-- ROS base_link to UR 'Base' Coordinates transform -->
    <link name="${prefix}base"/>
    <joint name="${prefix}base_link-base_fixed_joint" type="fixed">
    <!-- NOTE: this rotation is only needed as long as base_link itself is
         not corrected wrt the real robot (ie: rotated over 180
         degrees)
    -->
    <origin xyz="0 0 0" rpy="0 0 ${-pi}"/>
    <parent link="${prefix}base_link"/>
    <child link="${prefix}base"/>
    </joint>

    <!-- Frame coincident with all-zeros TCP on UR controller -->
    <link name="${prefix}tool0"/>
    <joint name="${prefix}wrist_3_link-tool0_fixed_joint" type="fixed">
    <origin xyz="0 ${wrist_3_length} 0" rpy="${pi/-2.0} 0 0"/>
    <parent link="${prefix}wrist_3_link"/>
    <child link="${prefix}tool0"/>
    </joint>

  </xacro:macro>
  </robot>
```

MoveIt Framework [1]

Move Group



The figure above shows the high-level system architecture for the primary node provided by MoveIt! called `Move_Group`. This node serves as an integrator: pulling all the individual components together to provide a set of ROS actions and services for users to use. `Move_Group` is a ROS node. It uses the ROS param server to get three kinds of information:

- URDF - `Move_Group` looks for the `robot_description` parameter on the ROS param server to get the URDF for the robot.

- SRDF - `Move_Group` looks for the `robot_description_semantic` parameter on the ROS param server to get the SRDF for the robot. The SRDF is typically created (once) by a user using the MoveIt! Setup Assistant.
- MoveIt! configuration - `Move_Group` will look on the ROS param server for other configuration specific to MoveIt! including joint limits, kinematics, motion planning, perception and other information. Config files for these components are automatically generated by the MoveIt! setup assistant and stored in the config directory of the corresponding MoveIt! config package for the robot.

Robot Interface

`move_group` talks to the robot through ROS topics and actions. It communicates with the robot to get current state information (positions of the joints, etc.), to get point clouds and other sensor data from the robot sensors and to talk to the controllers on the robot.

Joint State Information

`move_group` listens on the `/joint_states` topic for determining the current state information - i.e. determining where each joint of the robot is. `move_group` is capable of listening to multiple publishers on this topic even if they are publishing only partial information about the robot state (e.g. separate publishers may be used for the arm and mobile base of a robot). Note that `move_group` will not setup its own joint state publisher - this is something that has to be implemented on each robot.

Transform Information

`move_group` monitors transform information using the ROS TF library. This allows the node to get global information about the robot's pose (among other things). E.g., the ROS navigation stack will publish the transform between the map frame and base frame of the robot to TF. `move_group` can use TF to figure out this transform for internal use. Note that `move_group` only listens to TF. To publish TF information from your robot, you will need to have a `robot_state_publisher` node running on your robot.

Controller Interface

`move_group` talks to the controllers on the robot using the FollowJointTrajectoryAction interface. This is a ROS action interface. A server on the robot needs to service this action - this server is not provided by `move_group` itself. `move_group` will only instantiate a client to talk to this controller action server on your robot.

Planning Scene

`move_group` uses the Planning Scene Monitor to maintain a planning scene, which is a representation of the world and the current state of the robot. The robot state can include any objects carried by the robot which are considered to

be rigidly attached to the robot. More details on the architecture for maintaining and updating the planning scene are outlined in the Planning Scene section below.

Motion Planning

The Motion Planning Plugin

MoveIt! works with motion planners through a plugin interface. This allows MoveIt! to communicate with and use different motion planners from multiple libraries, making MoveIt! easily extensible. The interface to the motion planners is through a ROS Action or service (offered by the `move_group` node). The default motion planners for `move_group` are configured using OMPL and the MoveIt! interface to OMPL by the MoveIt! Setup Assistant.

The Motion Plan Request

The motion plan request clearly specifies what you would like the motion planner to do. Typically, you will be asking the motion planner to move an arm to a different location (in joint space) or the end-effector to a new pose. Collisions are checked for by default (including self-collisions). You can attach an object to the end-effector (or any part of the robot), e.g. if the robot picks up an object. This allows the motion planner to account for the motion of the object while planning paths. You can also specify constraints for the motion planner to check - the inbuilt constraints provided by MoveIt! are kinematic constraints:

- Position constraints - restrict the position of a link to lie within a region of space
- Orientation constraints - restrict the orientation of a link to lie within specified roll, pitch or yaw limits
- Visibility constraints - restrict a point on a link to lie within the visibility cone for a particular sensor
- Joint constraints - restrict a joint to lie between two values
- User-specified constraints - you can also specify your own constraints with a user-defined callback.

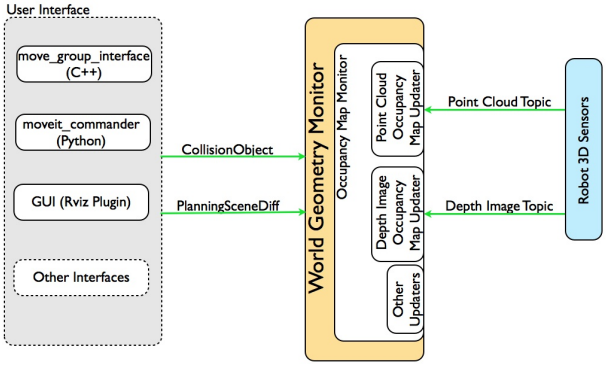
The Motion Plan Result

The movegroup node will generate a desired trajectory in response to your motion plan request. This trajectory will move the arm (or any group of joints) to the desired location. Note that the result coming out of `move_group` is a trajectory and not just a path - `_move_group` will use the desired maximum velocities and accelerations (if specified) to generate a trajectory that obeys velocity and acceleration constraints at the joint level.

OMPL

OMPL (Open Motion Planning Library) is an open-source motion planning library that primarily implements randomized motion planners. MoveIt! integrates directly with OMPL and uses the motion planners from that library as its primary/default set of planners. The planners in OMPL are abstract; i.e. OMPL has no concept of a robot. Instead, MoveIt! configures OMPL and provides the back-end for OMPL to work with problems in Robotics.

Planning Scene



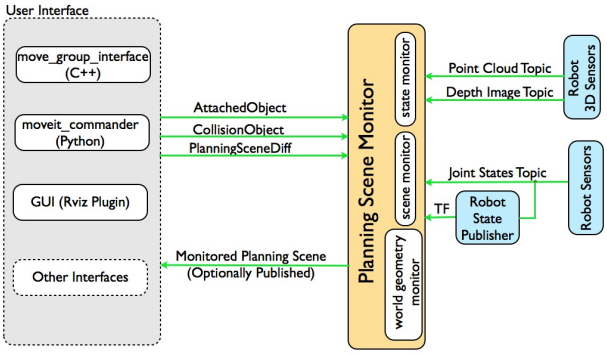
The planning scene is used to represent the world around the robot and also stores the state of the robot itself. It is maintained by the planning scene monitor inside the move group node. The planning scene monitor listens to:

- State Information: on the `joint_states` topic
- Sensor Information: using the world geometry monitor described below
- World geometry information: from user input on the `planning_scene` topic (as a planning scene diff).

World Geometry Monitor

The world geometry monitor builds world geometry using information from the sensors on the robot and from user input. It uses the occupancy map monitor described below to build a 3D representation of the environment around the robot and augments that with information on the `planning_scene` topic for adding object information.

3D Perception



3D perception in MoveIt! is handled by the occupancy map monitor. The occupancy map monitor uses a plugin architecture to handle different kinds of sensor input as shown in the Figure above. In particular, MoveIt! has inbuilt support for handling two kinds of inputs:

- Point clouds: handled by the point cloud occupancy map updater plugin
- Depth images: handled by the depth image occupancy map updater plugin

Note that you can add your own types of updaters as a plugin to the occupancy map monitor.

Octomap

The Occupancy map monitor uses an Octomap to maintain the occupancy map of the environment. The Octomap can actually encode probabilistic information about individual cells although this information is not currently used in MoveIt!. The Octomap can directly be passed into FCL, the collision checking library that MoveIt! uses.

Depth Image Occupancy Map Updater

The depth image occupancy map updater includes its own self-filter, i.e. it will remove visible parts of the robot from the depth map. It uses current information about the robot (the robot state) to carry out this operation.

Kinematics

Kinematics Plugin

MoveIt! uses a plugin infrastructure, especially targeted towards allowing users to write their own inverse kinematics algorithms. Forward kinematics and finding jacobians is integrated within the RobotState class itself. The default inverse kinematics plugin for MoveIt! is configured using the KDL numerical jacobian-based solver. This plugin is automatically configured by the MoveIt! Setup Assistant. It is also possible to implement yout own kinematics solvers, using the IKFast package to generate the C++ code needed to work with your particular robot.

Kinematics and Dynamics Library (KDL) [8]

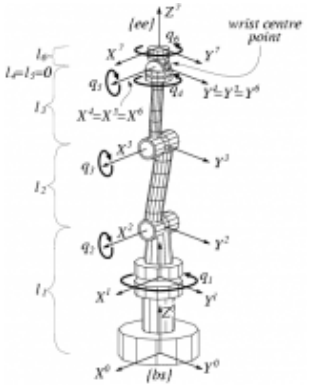


Figure 14: Kinematic structure example

Skeleton of a serial robot arm with six revolute joints. This is one example of a kinematic structure, reducing the motion modelling and specification to a geometric problem of relative motion of reference frames. The Kinematics and Dynamics Library (KDL) develops an application independent framework for modelling and computation of kinematic chains, such as robots, biomechanical human models, computer-animated figures, machine tools, etc. It provides class libraries for geometrical objects (point, frame, line,...), kinematic chains of various families (serial, humanoid, parallel, mobile,...), and their motion specification and interpolation.

Collision Checking

Collision checking in MoveIt! is configured inside a Planning Scene using the CollisionWorld object. Fortunately, MoveIt! is setup so that users never really have to worry about how collision checking is happening. Collision checking in MoveIt! is mainly carried out using the FCL package - MoveIt!'s primary CC library.

Collision Objects

MoveIt! supports collision checking for different types of objects including:

- Meshes
- Primitive Shapes - e.g. boxes, cylinders, cones, spheres and planes
- Octomap - the Octomap object can be directly used for collision checking

In the following figure 15 is represented the runtime situation in our environment, the robot using the cloud point is able to reconstruct the environment around creating a map of the obstacles. Thanks to this discrete(Voxel map) representation of the obstacles the robot has a basic knowledge of the environment, this representation is also affected by noise due to the noise received by the sensor as can be seen in 15.

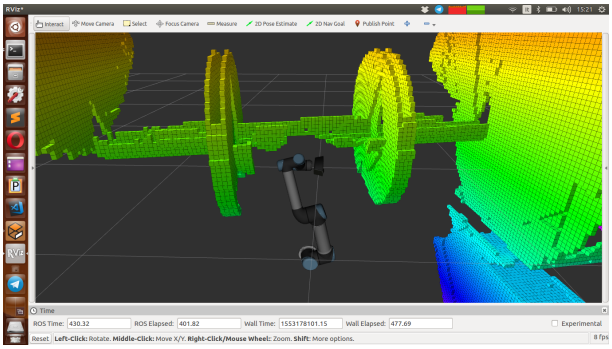


Figure 15: Octomap created by Voxels used for collision avoidance

PCL Sample consensus[2]

The `pcl_sample_consensus` library holds SAmple Consensus (SAC) methods like RANSAC and models like planes and cylinders. These can combined freely in order to detect specific models and their parameters in point clouds. Random sample consensus (RANSAC) is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. Therefore, it also can be interpreted as an outlier detection method. It is a non-deterministic algorithm in the sense that it produces a reasonable result only with a certain probability, with this probability increasing as more iterations are allowed. The RANSAC [6] algorithm is a learning technique to estimate parameters of a model by random sampling of observed data. Given a dataset whose data elements contain both inliers and outliers, RANSAC uses the voting scheme to find the optimal fitting result. Data elements in the dataset are used to vote for one or multiple models. The implementation of this voting scheme is based on two assumptions: that the noisy features will not vote consistently for any single model (few outliers) and there are enough features to agree on a good model (few missing data). The RANSAC algorithm is essentially composed of two steps that are iteratively repeated:

- In the first step, a sample subset containing minimal data items is randomly selected from the input dataset. A fitting model and the corresponding model parameters are computed using only the elements of this sample subset. The cardinality of the sample subset is the smallest sufficient to determine the model parameters.
- In the second step, the algorithm checks which elements of the entire dataset are consistent with the model instantiated by the estimated model parameters obtained from the first step. A data element will be considered as an outlier if it does not fit the fitting model instantiated by the set of estimated model parameters within some error threshold that defines the maximum deviation attributable to the effect of noise.

The set of inliers obtained for the fitting model is called consensus set. The RANSAC algorithm will iteratively repeat the above two steps until the obtained consensus set in certain iteration has enough inliers. The input to the RANSAC algorithm is a set of observed data values, a way of fitting some kind of model to the observations, and some confidence parameters. RANSAC achieves its goal by repeating the following steps:

- Select a random subset of the original data. Call this subset the hypothetical inliers.
- A model is fitted to the set of hypothetical inliers.
- All other data are then tested against the fitted model. Those points that fit the estimated model well, according to some model-specific loss function, are considered as part of the consensus set.
- The estimated model is reasonably good if sufficiently many points have been classified as part of the consensus set.

- Afterwards, the model may be improved by reestimating it using all members of the consensus set.

This procedure is repeated a fixed number of times, each time producing either a model which is rejected because too few points are part of the consensus set, or a refined model together with a corresponding consensus set size. In the latter case, we keep the refined model if its consensus set is larger than the previously saved model.

References

[1] MoveIt framework (`https://moveit.ros.org/`)

[2] Point Cloud Library (`http://www.pointclouds.org/`)

[3] Universal Robot (`https://www.universal-robots.com/it/`)

[4] Universal Robot ROS package (`http://wiki.ros.org/universal_robot`)

[5] A collection of commonly used sensors: urdf files and a few tools (`https://github.com/JenniferBuehler/common-sensors`)

[6] RANSAC algorithm to fit data (`https://en.wikipedia.org/wiki/Random_sample_consensus`)

[7] Rospy package (`http://wiki.ros.org/rospy`)

[8] Open Robot Control Software (`http://www.orocos.org/`)