Institute for Cognitive Systems
Technische Universität München
Prof. Gordon Cheng

Multi-sensory Based Robot Dynamic Manipulation
ROS Tutorial 3: 3D Robot Modeling with ROS

1 Modeling a robot on ROS

a Understanding robot modeling using URDF

ROS naturally comes with a wide variety of powerful tools, allowing to generate three-dimensional animated representations of robots. These representations are usually coded into so called "URDF" files. The Unified Robot Description Format (URDF) is a specific XML convention, describing robots in a way that is both both human- and machine-readable. A URDF file basically contains all the informations necessary to generate the *geometric, kinematic*, *dynamic*, *collision*, and *sensory* models of a specific robot. These informations are organized using a set of dedicated *tags*: since robots are defined as a set of links, joints and sensors, the corresponding tags allow to precisely parameterize each of these elements:

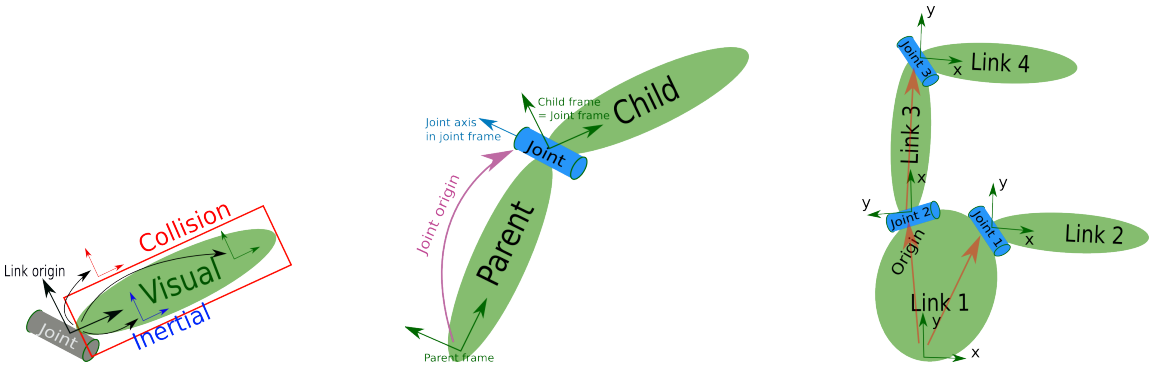
- **Link**: The link tag allows to model a robot link and its properties. The syntax is as follows:

```
<link name="<name_of_the_link>">
  <inertial>...........</inertial>        #Optional
  <visual> ............</visual>          #Optional
  <collision>..........</collision>       #Optional
</link>
```

  As indicated by its name, the "`visual`" section describes the link visual properties (e.g. size, geometry or color). It is even possible to import a 3D mesh in order to represent the robot link in a more realistic way. The "`collision`" section describes the link collision model. This model usually encapsulates the real link in order to detect collision before it actually happens. Finally the "`inertial`" section defines the link dynamic parameters (e.g. mass, inertia). Figure 1a shows a representation of a single link as it is defined in the URDF convention. More details can be found at the following address: `http://wiki.ros.org/urdf/XML/link`.

- **Joint**: The joint tag is used in order to describe robot joints and their properties. The syntax is as follows:

```
<joint name="name_of_the_joint" type="type_of_the_joint">
  <origin xyz="z y z" rpy="r p y"/>              #Optional
  <parent link="link1"/>                         #Required
  <child link="link2"/>                          #Required
  <axis .... />                                  #Optional
  <calibration rising, falling />                #Optional
  <dynamics damping, friction/>                  #Optional
  <limit lower, upper, effort, velovity />       #Optional
</joint>
```



(a) Visualization of a URDF link

(b) Visualization of a URDF joint

(c) Robot URDF model

Figure 1: URDF tags

A URDF joint is always defined with respect to a `parent` and a `child` link. The joint tag supports the following types of joints: prismatic, revolute, continuous, fixed, floating, and planar. Optionally, joints can be provided with dynamics properties, such as damping or friction. A set of limits can even be defined in order to better fit to the real robot. Figure 1b provides an illustration of a robot joint with its different links. More details can be found at the following address: `http://wiki.ros.org/urdf/XML/joint`.

- **Robot**: This tag describes the root element of the URDF: it must *encapsulate the entire robot model*. Inside the robot tag, we can define the name of the robot as well as its different links and joints. The syntax is as follows:

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

  As showed in Figure 1c, the robot model consists of a set of *links*, interconnected by *joints*, into a *kinematic chain*.

- More URDF tags can be found at `http://wiki.ros.org/urdf/XML`.

In ROS, the `urdf` package provides a powerful C++ parser, allowing to recursively generate the entire mathematical model of a robot based on its URDF file. Although highly versatile, the URDF is however limited to the description of rigid robots with tree-like[1] kinematic chains.

[1] Parallel robots cannot be described using URDF

b Understanding robot modeling using XACRO

b.1 Motivation

The use of URDF may become problematic in the case of complex robotic systems, since its non-modular nature unnecessarily increases the code complexity as the number of degrees of freedom of the considered robot get bigger. In fact, some of the main features that URDF is missing are the simplicity, reusability, modularity, and programmability. Modularity, in particular, refers to the possibility of including several URDF files as subparts of a main robot description file, thereby resulting in enhanced code readability. Programmability here refers to the possibility of defining variables, constants, mathematical expressions, or conditional statement, in the description language, in order to make it more user friendly. XACRO (Xml-mACROS) can be considered as an updated version of URDF, created with these problems in mind. Capable of generating – or importing – reusable macros within a given robot description, the XACRO language moreover support simple programming statements in its description. The possibility of using variables, constants, mathematical expressions or conditional statements makes the robot description more intelligent and efficient. XACRO files can be automatically converted to URDF whenever it is necessary, using dedicated ROS tools.

b.2 Using properties

Using XACRO, we can declare constants or properties which can be used anywhere in the code. The main use of these constant definitions are, instead of giving hard coded values on links and joints, we can keep constants like this and it will be easier to change these values rather than finding the hard coded values and replacing them. An example of using properties are given here. We declare the base link and pan link's length and radius. So, it will be easy to change the dimension here rather than changing values in each one:

```
<xacro:property name="base_link_length" value="0.01" />
<xacro:property name="base_link_radius" value="0.2" />
<xacro:property name="pan_link_length" value="0.4" />
<xacro:property name="pan_link_radius" value="0.04" />
```

We can use the value of the variable by replacing the hard coded value by the following definition as given here:

```
<cylinder length="${pan_link_length}"
radius="${pan_link_radius}"/>
```

Here, the old value "0.4" is replaced with "{pan_link_radius}" , and "0.04" is replaced with "{pan_link_radius}" .

b.3 Using the math expression

We can build mathematical expressions inside ${} using the basic operations such as + , - , * , / , unary minus, and parenthesis. Exponentiation and modulus are not supported yet. The following is a simple math expression used inside the code:

```
<cylinder length="${pan_link_length}"
radius="${pan_link_radius+0.02}"/>
```

b.4 Using macros

One of the main features of xacro is that it supports macros. We can reduce the length complex definition using xacro to a great extent. Here is a xacro definition we used in our code for inertial:

```
<xacro:macro name="inertial_matrix" params="mass">
<inertial>
<mass value="${mass}" />
<inertia ixx="0.5" ixy="0.0" ixz="0.0"
iyy="0.5" iyz="0.0" izz="0.5" />
</inertial>
</xacro:macro>
```

Here, the macro is named `inertial_matrix` , and its parameter is mass. The mass parameter can be used inside the inertial definition using ${mass} . We can replace each inertial code with a single line as given here:

```
<xacro:inertial_matrix mass="1"/>
```

The xacro definition improved the code readability and reduced the number of lines compared to urdf. Next, we can see how to convert xacro to the urdf file.

b.5 Conversion of XACRO to URDF

After designing the xacro file, we can use the following command to convert it into a UDRF file:

```
$ rosrun xacro xacro.py pan_tilt.xacro > pan_tilt_generated.urdf
```

We can use the following line in the ROS launch file for converting xacro to UDRF and use it as a `robot_description` parameter:

```
<param name="robot_description" command="$(find xacro)/xacro.py
$(find mastering_ros_robot_description_pkg)/urdf/pan_tilt.xacro"/>
```

2 Building a URDF robot model

1. Create a folder for your model inside your catkin workspace:

   ```
   $ mkdir urdf_tutorial
   ```

2. Create a ros package for the robot description:

   ```
   $ cd urdf_tutorial
   $ catkin_create_pkg robot0_description urdf
   ```

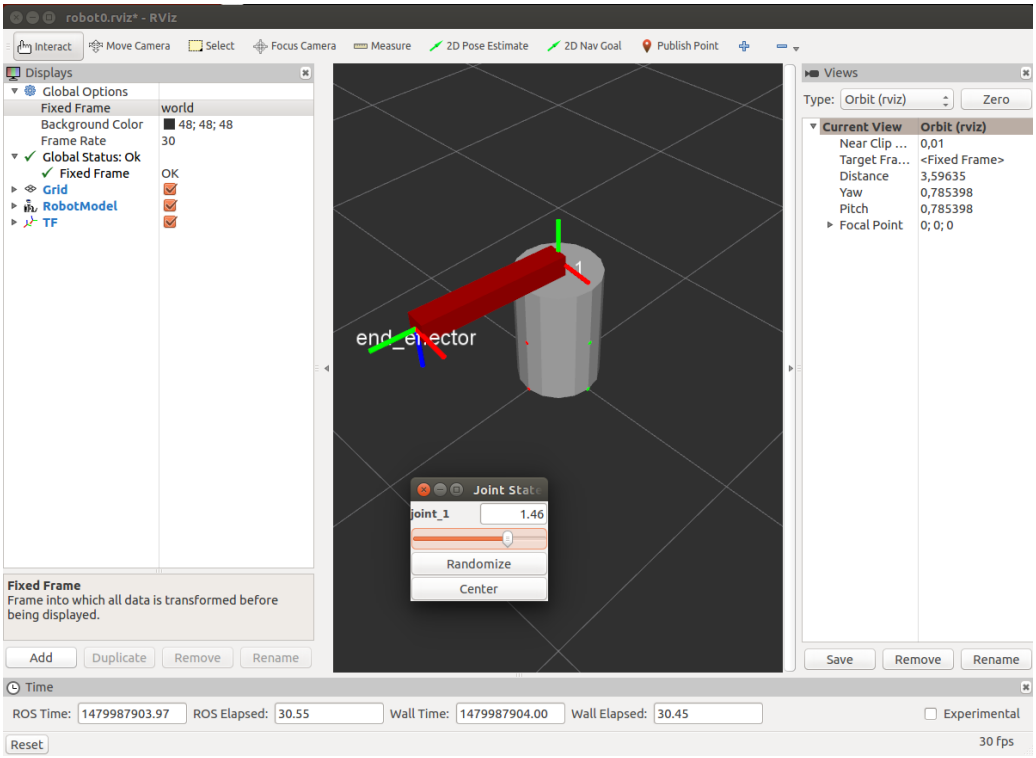


Figure 2: Robot visualization on Rviz

3. Create a folder called urdf:

```
$ cd robot0_description
$ mkdir urdf
```

4. Create a file named `robot0.urdf`:

```
$ cd urdf
$ gedit robot0.urdf
```

5. Start the file as a standard xml 1.0. Then start a robot element:

```
<?xml version="1.0"?>
<robot name="robot0">
</robot>
```

6. Define the first link as world. This will be an empty link and is used to state the world frame. Just after this link add a joint element as fixed.

```
<link name="world" />

<joint name="joint_0" type="fixed">
```

```
<parent link="world"/>
<child link="link_0"/>
</joint>
```

7. After this, we can define the further kinematic chain with as many links and joints as needed:

```
<?xml version="1.0"?>
<robot name="robot0">

    <link name="world" />

    <joint name="joint_0" type="fixed">
        <parent link="world"/>
        <child link="link_0"/>
    </joint>

    <link name="link_0">
        <visual>
            <geometry>
                <cylinder length="0.6" radius="0.2"/>
            </geometry>
            <material name="gray">
                <color rgba="0.5 0.5 0.5 1"/>
            </material>
        </visual>
    </link>

    <joint name="joint_1" type="revolute">
        <origin xyz="0 0 0.35" rpy="1.57079632679 0 0"/>
        <parent link="link_0"/>
        <child link="link_1"/>
        <limit effort="30" velocity="1.0" lower="-3.1415926535897931" upper
    ="3.1415926535897931" />
        <axis xyz="0 1 0"/>
    </joint>

    <link name="link_1">
        <visual>
            <origin xyz="0 0 0.35" rpy="0 0 0"/>
            <geometry>
                <box size="0.1 0.1 0.7" />
            </geometry>
            <material name="red">
                <color rgba="0.5 0.0 0.0 1"/>
            </material>
        </visual>
    </link>

    <joint name="end_efector_joint" type="fixed">
        <origin xyz="0 0 0.7" rpy="1.57079632679 0 0"/>
        <parent link="link_1"/>
```

```
        <child link="end_effector"/>
    </joint>

    <link name="end_effector" />

</robot>
```

8. To test the descriptor, create another package inside the `urdf_tutorial` folder.

```
$ catkin_create_pkg robot0_bringup robot_state_publisher robot0_description
```

9. Create a launch folder and a launch file inside this package:

```
$ cd robot0_bringup
$ mkdir launch
$ cd launch
$ gedit robot0_bringup.launch
```

10. Copy this code inside the launch file:

```
<?xml version="1.0"?>

<launch>

    <arg name="gui" default="false" />

    <param name="robot_description" command="cat $(find robot0_description)/urdf/
    robot0.urdf" />
    <param name="use_gui" value="$(arg gui)"/>

    <node name="joint_state_publisher" pkg="joint_state_publisher" type="
    joint_state_publisher" />

    <node name="robot_state_publisher" pkg="robot_state_publisher" type="
    state_publisher" />
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find robot0_bringup)/launch
    /rviz_config/robot0.rviz" required="true" />

</launch>
```

11. Compile the catkin workspace.
12. Run in one terminal the next command to test the model with a joint command gui:

```
$ roslaunch robot0_bringup robot0_bringup.launch gui:=true
```

13. The rviz window will look empty. Here set the fixed frame to world and import the robot model and the TF tree.
14. Save the rviz config file into the launch file in a folder named `rviz_config` as `robot0.rviz`

3 Building a XACRO model using macros

1. Create a ros package for the robot description:

```
$ cd urdf_tutorial
$ catkin_create_pkg r1_robot_description urdf xacro
```

2. Create a folder called urdf:

```
$ cd r1_robot_description
$ mkdir urdf
```

3. Create a file named `r1_robot.xacro`:

```
$ cd urdf
$ gedit r1_robot.xacro
```

4. Start the file as a standard xml 1.0. Then start a robot element stating the xacro specification for the parser. In this case we will not specify the robot name:

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

</robot>
```

5. Inside the robot element, we can define constants as properties:

```
<property name="M_PI" value="3.1415926535897931" />
<property name="DEG2RAD" value="0.01745329251994329577" />
```

6. After this, we can define the xacro macro specifying the arguments. Note that the arguments are used inside the macro using the `${ }` tags:

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

    <property name="M_PI" value="3.1415926535897931" />
    <property name="DEG2RAD" value="0.01745329251994329577" />

    <xacro:macro name="single_joint_robot" params="name xposition yposition">

    <joint name="${name}_joint_0" type="fixed">
        <parent link="world"/>
        <child link="${name}_link_0"/>
        <origin rpy="0 0 0" xyz="${xposition} ${yposition} 0" />
    </joint>

    <link name="${name}_link_0">
        <visual>
            <geometry>
                <cylinder length="0.6" radius="0.2"/>
            </geometry>
```

```
            <material name="gray">
                <color rgba="0.5 0.5 0.5 1"/>
            </material>
        </visual>
    </link>

    <joint name="${name}_joint_1" type="revolute">
        <origin xyz="0 0 0.35" rpy="${M_PI/2} 0 0"/>
        <parent link="${name}_link_0"/>
        <child link="${name}_link_1"/>
        <limit effort="30" velocity="1.0" lower="-${M_PI}" upper="${M_PI}" />
        <axis xyz="0 1 0"/>
    </joint>

    <link name="${name}_link_1">
        <visual>
            <origin xyz="0 0 0.35" rpy="0 0 0"/>
            <geometry>
                <box size="0.1 0.1 0.7" />
            </geometry>
            <material name="red">
                <color rgba="0.5 0.0 0.0 1"/>
            </material>
        </visual>
    </link>

    <joint name="${name}_end_efector_joint" type="fixed">
        <origin xyz="0 0 0.7" rpy="${M_PI/2} 0 0"/>
        <parent link="${name}_link_1"/>
        <child link="${name}_end_effector"/>
    </joint>

    <link name="${name}_end_effector" />

    </xacro:macro>

</robot>
```

7. Now create a folder named `robots` inside the `r1_robot_description` package

```
$ cd r1_robot_description
$ mkdir robots
```

8. Create a file named `2robot.xacro`:

```
$ cd robots
$ gedit 2robot.xacro
```

9. In this file we will start a robot model specifying a name and the xacro parser. Then include the `r1_robot.xacro`. Finally, create a $world$ link and use the `single_joint_robot` macro twice to create two ramifications from the world link.

```
<?xml version="1.0"?>

<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="r1_robot">

    <xacro:include filename="$(find r1_robot_description)/urdf/r1_robot.xacro" />

    <link name="world" />

    <xacro:single_joint_robot name= "robot1" xposition="0.0" yposition="0.0" />

    <xacro:single_joint_robot name= "robot2" xposition="1.0" yposition="0.0" />

</robot>
```

10. To test the descriptor, create another package inside the `urdf_tutorial` folder.

```
$ catkin_create_pkg r1_robot_bringup robot_state_publisher r1_robot_description
```

11. Create a launch folder and a launch file inside this package:

```
$ cd r1_robot_bringup
$ mkdir launch
$ cd launch
$ gedit two_robots_bringup.launch
```

12. Copy this code inside the launch file:

```
<?xml version="1.0"?>

<launch>

    <arg name="gui" default="false" />

    <param name="robot_description" command="$(find xacro)/xacro.py '$(find
    r1_robot_description)/robots/2robot.xacro'" />
    <param name="use_gui" value="$(arg gui)"/>

    <node name="joint_state_publisher" pkg="joint_state_publisher" type="
    joint_state_publisher" />

    <node name="robot_state_publisher" pkg="robot_state_publisher" type="
    state_publisher" />
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(find r1_robot_bringup)/
    launch/rviz_config/r1_robot.rviz" required="true" />

</launch>
```

Note that this time, the `robot_description` is loaded using the xacro parser.

13. Compile the catkin workspace.
14. Run in one terminal the next command to test the model with a joint command gui:

```
$ roslaunch r1_robot_bringup r1_robot_bringup.launch gui:=true
```

15. The rviz window will look empty. Here set the fixed frame to world and import the robot model and the TF tree.
16. Save the rviz config file into the launch file in a folder named `rviz_config` as `r1_robot.rviz`

4 Homework

a Building a 4DOF realistic robot model

Now the interesting things begin:

1. Use the meshes in the `rppr_robot_meshes` folder to build the first robot model from the DH tutorial (Tutorial 2). **Important note**: To import a mesh inside a geometry element use:

```
<geometry>
  <mesh filename="package://rppr_robot_description/meshes/XXX.stl"/>
</geometry>
```

2. Do you have to follow the DH convention to build a URDF file?
3. What is the difference between a URDF file and a XACRO file?
4. What are empty links used for?
5. Compare the end effector position in rviz with the implementation in Matlab.

b Delivery format and due date

You should deliver a compressed "`.zip`" file with your implementations using the instructions of the exercise sheet and of the given template. Include a "`Readme.txt`" file to indicate how to run your nodes and to answer the questions. Please, name the compressed file as follows: "`Name_lastName_MSRDM_ROStutorial3.zip`'. This tutorial will have to be delivered no later than the **Monday 05/12/2017, 23h59**.