Urdf Tutorial

① Building a Visual robot model with URDF from Scratch

<?xml version = "1.0"?>

<robot name = "~~my~~ Name of Robot">

{ Material defination }

{ Link 1 defination }

{ joint 1 defination }

{ Link 2 defination }

{ joint 2 defination }

⋮ ⋮ ⋮

</robot>

★ Material defination

<material name = "Name of material>

<color rgba = "0 0 0.8 1"/>

</material>

* Link defination

<link name="Name of the link">

<Visual>

<geometry>

{geometry definition}

</geometry>

<origin rpy="0.0 0" xyz "0 0 0"/>

<material name="Name of material to apply"/>

</Visual>

</link>

{Controls position and orientation of mesh relative to Origin}

Geometry defination

<cylinder length="0.6" radius="0.2"/>

<box size="0.6 0.1 0.2"/>

<sphere radius="0.2"/>

{For loading the basic shaps}

<mesh filename="Package://urdf_tutorial/meshes/1-finger.dae"/>

{For loading the mesh file}

* Joint defination

```
<joint name="name of joint" type="fixed">
    <Parent link="parent link name"/>
    <Child link="Child link name"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
</joint>
```

Controls position and Orientation of Child link wrt Parent link

Stl → Only mesh data.
dae → mesh + material data

② Building a Movable Robot model with URDF

Joint types ⇒ Continuous, revolute and prismatic.

Can take any angle from $-\infty$ to $+\infty$

<axis xyz="0 0 1"/>

Axis of rotation

They rotate in the same way that the continuous joints do, but they have strict limits

<axis xyz="0 0 1"/>

<limit effort="1000" lower="0.0" upper="0.548" velocity="0.5"/>

To limit effort (torque), rotation, angular velocity

Prismatic

<limit effor="1000" lower="-0.38" Upper="0" velocity="0.5"/>

* Other types of joint

Planar joint

Floating joint

③ Adding physical and Collision properties to a Urdf modle

★ Collision

⇒ Two main reason to make different visual and collision model :-

① Quicker processing

② Safe zone around the robot.

★ Physical properties

① Inertia

Kg

```
<inertial>
    <mass value="10"/>
    <inertia ixx="0.4" ixy="0.0" iyy="0.4"
             iyy="0.4" iyz="0.0" izz="0.2"/>
</inertial>
```

⇒ You can also specify an origin tag to specify the center of gravity and the inertial reference frame (relative to the link's reference frame).

(2) Contact coefficients

⇒ This is done with a subelement of the collision tag called contact_coefficients.

mu ⇒ friction coefficient
Ke ⇒ Stiffness coefficient
Kd ⇒ Dampening coefficient.

(3) Joint Dynamics

⇒ How the joint moves is defined by the dynamics tag for the joint.

↳ There are two attributes here:-

friction ⇒ Static friction

damping ⇒ physical damping value.

Scanned by CamScanner

(7) Using xacro to clean up a URDF file

⇒ Xacro package does three things that are very helpful.

(i) Constants
(ii) Simple math
(iii) Macros

⇒ xacro is a macro language.

★ **Constants**

{Name given to the Name Space}

<xacro: property name="width" value="0.2"/>

{defining the variable width with value 0.2}

⇒ Using that Value → "${width}"

★ **Simple math**

"${5/6}" ⟹ "0.8333..."

* Macros

Simple macro

```
<xacro:macro name="default_origin">
    <origin xyz="0 0 0" rpy="0 0 0"/>
</xacro:macro>
```

{using it}

```
<xacro:default_origin/>
```

Parameterized macro

```
<xacro:macro name="default_inertial"
                    params="mass">
  <inertial>
      <mass value="${mass}"/>
      <inertia .... />
  </inertial>
</xacro:macro>
```

{using it}

```
<xacro:default_inertial mass="10"/>
```

(5) Using a URDF in Gazebo

① Gazebo plugin

⇒ To get ROS to interact with Gazebo, we have to dynamically link to the ROS library that will tell Gazebo what to do.

⇒ To link Gazebo and ROS, we specify the plugin in the URDF, right before closing </robot> tag.

```
<gazebo>
  <plugin name="gazebo_ros_control"
          filename="libgazebo_ros_control.so">
    <robotNamespace>/</robotNamespace>
  </plugin>
</gazebo>
```

② Transmission

⇒ For every non fixed joint, we need to specify a transmission, which tells Gazebo what to do with the joint.

Example

```
<transmission name="head_swivel_trans">
    <type>transmission_interface/SimpleTransmission
                                        </type>
    <actuator name="$head_swivel_motor">
        <mechanicalReduction>1</mechanicalReduction>
    </actuator>
    <joint name="head_swivel">
        <hardwareInterface>PositionJointInterface
                    </hardwareInterface>
    </joint>
</transmission>
```

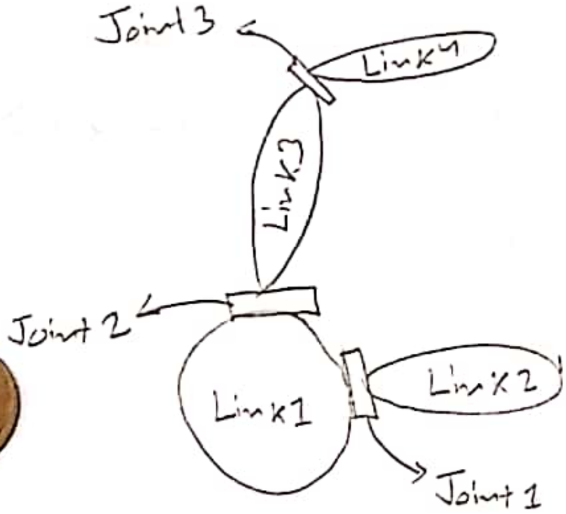
⇒ Just treat most of this chunk of code as boilerplate.

{ refers to section of code that have to be included in many places with little or no attention }

{ This should match the joint name }

Learning URDF (including C++ API)

1. Create your own urdf file



Parsing

It is ~~the~~ the process of analyzing text made of a sequence of tokens to determine its grammatical structure with respect to a given formal grammar.

⇒ There is a simple command line tool that will parse a urdf file for you, and tell you if ~~yo~~ the syntax is correct.

↳ liburdfdom-tool

→ sudo apt-get install liburdfdom-tool

⇒ Using the tools:

{check_urdf my_robot.urdf}

Scanned by CamScanner

⇒ You can visulize the URDF using graphiz.

urdf_to_graphiz myrobot.urdf

→ generates pdf

2. Parse a urdf file

⇒ Convention of storing robot in package name:

```
MYROBOT_description
 |→ Package.xml
 |→ CMakeLists.txt
 |→ /urdf
 |→ /meshes
 |→ /materials
 └→ /cad.
```

⇒ You can use urdf package to parse your urdf file

```
urdf :: Model model;
model.initFile(urdf_file)
```

{True if successfully Parsed} {False if not}

3. Using the robot State Publisher on your own robot

⇒ When you are working with a robot that has many relevant frames, it becomes quite a task to publish them all to tf.

↳ The robot State publisher is a tool that will do this job for you.

The robot State publisher helps you to broadcast the State of your robot to the tf transform library.

(a) Running as a node

Easiest way to run the robot state publisher is as a node.

⇒ You need two things to run the robot state publisher:

(i) Urdf xml robot description loaded on the parameter Server.

(ii) A Source that publishes the joint positions as a Sensor_msgs/JointState.

(5) Running as a library

⇒ Advanced user can also run the robot state publisher as a library, from within their own C++ code.

```
#include <robot_state_publisher/robot_state_publisher.h>
```

```
RobotStatePublisher(const KDL::Tree& tree);
```

{Constructor which takes KDL tree}

⇒ Now every time you want to publish the state of your robot, you call the PublishTransforms function:

```
Void publishTransforms(const std::map<std::string, double>& joint_positions, const ros::Time& time);
```

4. Start using the KDL parser

(a) Building the KDL parser

```
rosdep install kdl_parser
rosmake kdl_parser
```

(b) Using in your code

⇒ First add the KDL parser as a dependency to your package.xml.

```
<build_depend package="kdl_parser"/>
<run_depend package="kdl_parser"/>
```

⇒ To start using the KDL parser in your C++ code, include the following file:

```
#include <kdl_parser/kdl_parser.hpp>
```

⇒ Now there are different ways to proceed. You can construct a KDL tree from a urdf in various forms.

- → From a file
- → From a parameter server
- → From a xml element
- → From a urdf model

5. Using urdf with robot_state_publisher

→ First, we create the URDF model with all the necessary parts.

→ Then we write a node which publishes the JointState and transforms.

→ Finally we run all the parts together.

Robot model

⇒ All the robot model related files are kept in a package with name robotname_description
→ General convention

⇒ General directory tree of robotname_description:

robotname_description
- → CMakeLists.txt
- → package.xml

{The usual stuff}

- → meshes

{This directory is further subdivided into different directories according to different sub assemblies or sub categories, that contains mesh files (Stl, dae)}

Stl → {Only mesh data (binary)}

dae → {mesh + Color data}

Example
- → base
- → head
- → gripper
- → Sensor

- → materials
  - → texture

{Contains textures that may be applied}

*

- Urdf
  - This directories is further subdivided into different directories according to different sub assemblies or sub categories
  - Sub category1_V0 → Version number
  - Sub category2_V0
    - Sub category will be replaced by suitable name
  - ⋮
  - Sub category n_V0
  - common.xacro
    - This contains macros described that are common to different subdirectories
  - materials.urdf.xacro
    - This contains description of all the material

*

Sub category n_V0

- filename.gazebo.xacro
  - This contains gazebo related codes
- filename.transmission.xacro
  - This contains transmission related codes
- filename.urdf.xacro.
  - This is the complete xacro for the file which uses above two files to make it complete.

*

→ **robots**

{ This directory uses all the xacro defined in urdf directory to create different assemblies of robots, with different features. }

→ **gazebo**

↳ gazebo.urdf.xacro {directory}

{ This file is used by files in robot to add gazebo Plugin and add basic Settings }

→ **test**

{ This contains test files to ensure all the files work together Perfectly. }

→ **documents**

{ This generally contains .xls files , diagrams etc. }

{ For all Engineering drawings }

{ for all the Values }