Warning

You're reading the documentation for a version of ROS 2 that has reached its EOL (end-of-life), and is no longer officially supported. If you want up-to-date information, please have a look at Kilted.

Using Xacro to clean up your code

Goal: Learn some tricks to reduce the amount of code in a URDF file using Xacro

Tutorial level: Intermediate

Time: 20 minutes

Contents

By now, if you're following all these steps at home with your own robot design, you might be sick of doing all sorts of math to get very simple robot descriptions to parse correctly. Fortunately, you can use the xacro package to make your life simpler. It does three things that are very helpful.

- Constants
- Simple Math
- Macros

In this tutorial, we take a look at all these shortcuts to help reduce the overall size of the URDF file and make it easier to read and maintain.

Using Xacro

As its name implies, xacro is a macro language for XML. The xacro program runs all of the macros and outputs the result. Typical usage looks something like this:

```
xacro model.xacro > model.urdf
```

You can also automatically generate the urdf in a launch file. This is convenient because it stays up to date and doesn't use up hard drive space. However, it does take time to generate, so be aware that your launch file might take longer to start up.

```
path_to_urdf = get_package_share_path('pr2_description') / 'robots' / 'pr2.urdf.xacro'
robot_state_publisher_node = launch_ros.actions.Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    parameters=[{
        'robot_description': ParameterValue(
            Command(['xacro ', str(path_to_urdf)]), value_type=str
        )
    }]
)
```

At the top of the URDF file, you must specify a namespace in order for the file to parse properly. For example, these are the first two lines of a valid xacro file:

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

Constants

Let's take a quick look at our base_link in R2D2.

```
<link name="base_link">
  <visual>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
    <material name="blue"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.6" radius="0.2"/>
    </geometry>
  </collision>
</link>
```

The information here is a little redundant. We specify the length and radius of the cylinder twice. Worse, if we want to change that, we need to do so in two different places.

Fortunately, xacro allows you to specify properties which act as constants. Instead, of the above code, we can write this.

```
<xacro:property name="width" value="0.2" />
<xacro:property name="bodylen" value="0.6" />
<link name="base_link">
    <visual>
        <geometry>
            <cylinder radius="${width}" length="${bodylen}"/>
        </geometry>
        <material name="blue"/>
    </visual>
    <collision>
        <geometry>
            <cylinder radius="${width}" length="${bodylen}"/>
        </geometry>
    </collision>
</link>
```

- The two values are specified in the first two lines. They can be defined just about anywhere (assuming valid XML), at any level, before or after they are used. Usually they go at the top.
- Instead of specifying the actual radius in the geometry element, we use a dollar sign and curly brackets to signify the value.
- This code will generate the same code shown above.

The value of the contents of the ${} construct are then used to replace the ${}. This means you can combine it with other text in the attribute.

```
<xacro:property name=”robotname” value=”marvin” />
<link name=”${robotname}s_leg” />
```

This will generate

```
<link name=”marvins_leg” />
```

However, the contents in the ${} don’t have to only be a property, which brings us to our next point…

Math

You can build up arbitrarily complex expressions in the ${} construct using the four basic operations (+,-,*,/), the unary minus, and parenthesis. Examples:

```
<cylinder radius="${wheeldiam/2}" length="0.1"/>
<origin xyz="${reflect*(width+.02)} 0 0.25" />
```

You can also use more than the basic mathematical operations, like `sin` and `cos`.

Macros

Here’s the biggest and most useful component to the xacro package.

Simple Macro

Let’s take a look at a simple useless macro.

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

(This is useless, since if the origin is not specified, it has the same value as this.) This code will generate the following.

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

- The name is not technically a required element, but you need to specify it to be able to use it.
- Every instance of the `<xacro:$NAME />` is replaced with the contents of the `xacro:macro` tag.
- Note that even though its not exactly the same (the two attributes have switched order), the generated XML is equivalent.
- If the xacro with a specified name is not found, it will not be expanded and will NOT generate an error.

Parameterized Macro

You can also parameterize macros so that they don't generate the same exact text every time. When combined with the math functionality, this is even more powerful.

First, let's take an example of a simple macro used in R2D2.

```xml
<xacro:macro name="default_inertial" params="mass">
    <inertial>
            <mass value="${mass}" />
            <inertia ixx="1e-3" ixy="0.0" ixz="0.0"
                 iyy="1e-3" iyz="0.0"
                 izz="1e-3" />
    </inertial>
</xacro:macro>
```

This can be used with the code

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

The parameters act just like properties, and you can use them in expressions

You can also use entire blocks as parameters too.

```
<xacro:macro name="blue_shape" params="name *shape">
    <link name="${name}">
        <visual>
            <geometry>
                <xacro:insert_block name="shape" />
            </geometry>
            <material name="blue"/>
        </visual>
        <collision>
            <geometry>
                <xacro:insert_block name="shape" />
            </geometry>
        </collision>
    </link>
</xacro:macro>

<xacro:blue_shape name="base_link">
    <cylinder radius=".42" length=".01" />
</xacro:blue_shape>
```

- To specify a block parameter, include an asterisk before its parameter name.
- A block can be inserted using the insert_block command
- Insert the block as many times as you wish.

Practical Usage

The xacro language is rather flexible in what it allows you to do. Here are a few useful ways that xacro is used in the R2D2 model, in addition to the default inertial macro shown above.

To see the model generated by a xacro file, run the same command as with previous tutorials:

```
ros2 launch urdf_tutorial display.launch.py model:=urdf/08-macroed.urdf.xacro
```

(The launch file has been running the xacro command this whole time, but since there were no macros to expand, it didn't matter)

Leg macro

Often you want to create multiple similar looking objects in different locations. You can use a macro and some simple math to reduce the amount of code you have to write, like we do with R2's two legs.

```
<xacro:macro name="leg" params="prefix reflect">
    <link name="${prefix}_leg">
        <visual>
            <geometry>
                <box size="${leglen} 0.1 0.2"/>
            </geometry>
            <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
            <material name="white"/>
        </visual>
        <collision>
            <geometry>
                <box size="${leglen} 0.1 0.2"/>
            </geometry>
            <origin xyz="0 0 -${leglen/2}" rpy="0 ${pi/2} 0"/>
        </collision>
        <xacro:default_inertial mass="10"/>
    </link>

    <joint name="base_to_${prefix}_leg" type="fixed">
        <parent link="base_link"/>
        <child link="${prefix}_leg"/>
        <origin xyz="0 ${reflect*(width+.02)} 0.25" />
    </joint>
    <!-- A bunch of stuff cut -->
</xacro:macro>
<xacro:leg prefix="right" reflect="1" />
<xacro:leg prefix="left" reflect="-1" />
```

- Common Trick 1: Use a name prefix to get two similarly named objects.
- Common Trick 2: Use math to calculate joint origins. In the case that you change the size of your robot, changing a property with some math to calculate the joint offset will save a lot of trouble.
- Common Trick 3: Using a reflect parameter, and setting it to 1 or -1. See how we use the reflect parameter to put the legs on either side of the body in the base_to_${prefix}_leg origin.