Step 2: Using XACRO Files to Clean up URDF Files

From the previous tutorial, you learned how to manually create URDF files. However, there are a few major disadvantages to creating URDF files by hand:

- URDF files can get very long
- URDF files have repetitive code
- Constants, parameters, and arguments are not permitted
- Lacks mathematical operations
- Difficult and tedious to change robot dimensions

The `xacro` tool (pronounced "zacro", stands for XML Macro) was created to help with these shortcomings. We will learn how to use `.xacro` files to create our rrbot and table again. There are two very good resources for learning `xacro` on github or on the offical ros wiki. There are many features of `.xacro` files that we will not go into, such as adding conditional logic, loops, and embedding Python code, but you are always welcome to learn more with the links above!

Running an Example

From the `Step2` folder, run:

```
ros2 launch rrbot.launch.py
```

An RViz window and `joint_state_publisher_gui` should appear, and you should see two different rrbots. So what are the advantages of using `.xacro`? First, the physical dimensions of the robot have been completely parameterized using constants and mathematical expressions. Go into the `rrbot_macro.xacro` file. You will see many properties:

```
<xacro:property name="width" value="0.05" />
<xacro:property name="height0" value="2" /> <!-- base_link -->
<xacro:property name="height1" value="1" /> <!-- link1 -->
<xacro:property name="height2" value="1" /> <!-- link2 -->
<xacro:property name="axel_offset" value="0.05" />
```

You should play around with these values, and then run the launch script again. Just like magic, the robot has changed! This is one reason why URDF files are almost never written by hand - `xacro` just makes life easy!

Second, we can use xacro files to reuse URDF code. With just a few lines of code, we created a second rrbot and added it to RViz!

XACRO Summary

Here is a summary of what you can do with `.xacro`:

1. Define constants with `<xacro:property>`

2. Define arguments with `<xacro:arg>`
3. Include other files with `<xacro:include>`
4. Use mathematical operations such as `${width/2}` or `${length*cos(pi/3)}` or `${degrees(180)}`
5. Create custom macros which generates `.urdf` code for us, and allow us to reuse code.

rrbot XACRO Walkthrough

First, open the `rrbot.urdf.xacro` file. This is our "top-level" file, which is called by the `xacro` tool to generate a URDF file. By convention, the top-level file will end with `.urdf.xacro`.

```
<?xml version="1.0" encoding="UTF-8"?>
<robot xmlns:xacro="http://wiki.ros.org/xacro" name="rrbot">

  <!-- copy and paste the contents these .xacro files here -->
  <xacro:include filename="materials.xacro" />
  <xacro:include filename="rrbot_macro.xacro"/>

  <!-- create link fixed to the "world" -->
  <link name="world" />

  <!-- create the first robot -->
  <xacro:rrbot prefix="rrbot1_" parent="world">
    <!-- position robot in the world -->
    <origin xyz="0 0 0" rpy="0 0 0" />
  </xacro:rrbot>

  <!-- create the second robot -->
  <xacro:rrbot prefix="rrbot2_" parent="world">
    <!-- position robot in the world -->
    <origin xyz="3 0 0" rpy="0 0 ${degrees(180)}" />
  </xacro:rrbot>

</robot>
```

First, we see that the `<robot>` tag has the attribute `xmlns:xacro="http://www.ros.org/wiki/xacro"`. This is required at the start of every `.xacro` file. The `<xacro:include>` tags copy and paste the contents of `rrbot_macro.xacro` and `materials.xacro` into our top level file. Then the `world` link is defined. Finally, the `<xacro:rrbot>` macro is called, which creates the rrbot with the specified `parent` parameter and specified `origin`. A parameter called `prefix` is also defined, which we can use to add a prefix to every link and joint in the robot. This is necessary if there are two or more instances of a robot (every joint and link in a URDF file must have a unique name).

The `<xacro:rrbot>` macro is defined in `rrbot_macro.xacro`, which you should open now.

First, we see one very large macro called `rrbot`, which makes up the majority of the file. The parameters `prefix` and `parent` expect strings, while the `*origin` is a special type of parameter which passes an entire block of `urdf` code (in this case, it will just be the `<origin>` tag).

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
  <xacro:macro name="rrbot" params="prefix parent *origin">
```

Next, some properties for the dimensions of the rrbot are hard-coded as properties. (These values could also be read from a file, or passed in as parameters, but we will leave that for later.)

```
    <!-- Constants for robot dimensions -->
    <xacro:property name="width" value="0.05" /> <!-- Square dimensions
(widthxwidth) of beams -->
    <xacro:property name="height0" value="2" /> <!-- base_link -->
    <xacro:property name="height1" value="1" /> <!-- link1 -->
    <xacro:property name="height2" value="1" /> <!-- link2 -->
    <xacro:property name="axel_offset" value="0.05" /> <!-- Space between
top of beam and each joint -->
```

Then, the `base_joint` is defined:

```
    <joint name="${prefix}base_joint" type="fixed">
        <xacro:insert_block name="origin" />
        <parent link="${parent}" />
        <child link="${prefix}base_link" />
    </joint>
```

We see that the `<origin>` block from the `rrbot.urdf.xacro` file is copied into this joint (that is why we used `*origin` as our parameter). Also, the parent is set to the value of the parameter `${parent}`, which in this case is `world`.

The base link includes mathematical expressions inside a `${ }` block.

```
    <link name="${prefix}base_link">
        <visual>
        <origin xyz="0 0 ${height0/2}" rpy="0 0 0" />
        <geometry>
            <box size="${width} ${width} ${height0}" />
        </geometry>
        <material name="blue" />
        </visual>
    </link>
```

The `<joint1>` tag is shown below, as you would expect.

```
    <joint name="${prefix}joint1" type="continuous">
        <parent link="${prefix}base_link" />
        <child link="${prefix}link1" />
        <origin xyz="0 ${width} ${height0 - axel_offset}" rpy="0 0 0" />
        <axis xyz="0 1 0" />
    </joint>
```

The rest of the macro follows in a similar manner.

Launch File

The `xacro` tool can be used from the command line to generate a URDF file. For example,

```
xacro rrbot.urdf.xacro > rrbot.urdf
```

will create a pure URDF file which we can run with a launch script. However, we can skip this step by having our launch file create the URDF file *in memory* without actually writing to a new file. Take a look at `rrbot.launch.py`:

```
robot_description_content = Command(
    [
        PathJoinSubstitution([FindExecutable(name="xacro")]),
        " ",
        "rrbot.urdf.xacro",
    ]
)

robot_description = {"robot_description": robot_description_content}
```

This code is responsible for calling `xacro rrbot.urdf.xacro` and then passing the text to `robot_description`, which is then used when running the `robot_state_publisher` node. Even though we are not using the command line version of `xacro` to write urdf code to a file, it is a useful trick when trying to debug your macros. (Try just running `xacro rrbot.urdf.xacro` on your terminal and seeing the results!)

Task 1: Create a Four-Legged Table

Using what you have learned about `xacro`, recreate your four-legged rectangular table.

- You should create only three files: `table.urdf.xacro` and `table_macro.xacro` and `table.launch.py`.
- You should create the following xacro properties, initially:
  - `table_length = 2`
  - `table_width = 1`

    - `table_height = 1`
    - `table_thickness = 0.05`
    - `leg_length = ${table_height - table_thickness}`
    - `leg_radius = 0.05`
- All links should have the proper visual properties. Use a `brown` color for the table top and a `light_grey` color for the legs.

Once you have a table that looks just like the one you made in Step 1, **change the `table_length` and `table_width` properties to your own custom values**, and verify that the legs remain tangent to the sides of the table. Take a screenshot of your resized table in the RViz window for your lab report. Ensure the world grid is touching the bottom of the table legs.

Task 2: Mount the robot on the table.

Create a launch script called `rrbot_on_table.launch.py` that visualizes the robot on the the table. The `rrbot_on_table.urdf.xacro` file is provided for you. (For the rest of this lab, you are free to make the table dimensions anything you'd like, including the defaults provided in the previous task.)

Take a screenshot of the RViz window for your lab report.

Task 3: Table on RRBot

In a similar fashion, create a launch script for the provided `table_on_rrbot.urdf.xacro` file which mounts the child `top` frame to parent `tool0`.

Take a screenshot of the RViz window for your lab report.

Task 4: Robotics Lab

Create a robotics laboratory in RViz, by placing multiple robots and tables in the room. You should have at least three tables, each with one or more rrbot arms mounted on them. Answers will vary. (You need to just create another `.urdf.xacro` file and corresponding launch script. Mount the robots on the tables as you wish). The point of this task is to get comfortable with using xacro macros multiple times with different arguments, and to see the importance of the `prefix` parameter.

Take a screenshot of the RViz window.

Next Steps

Proceed to Step 3