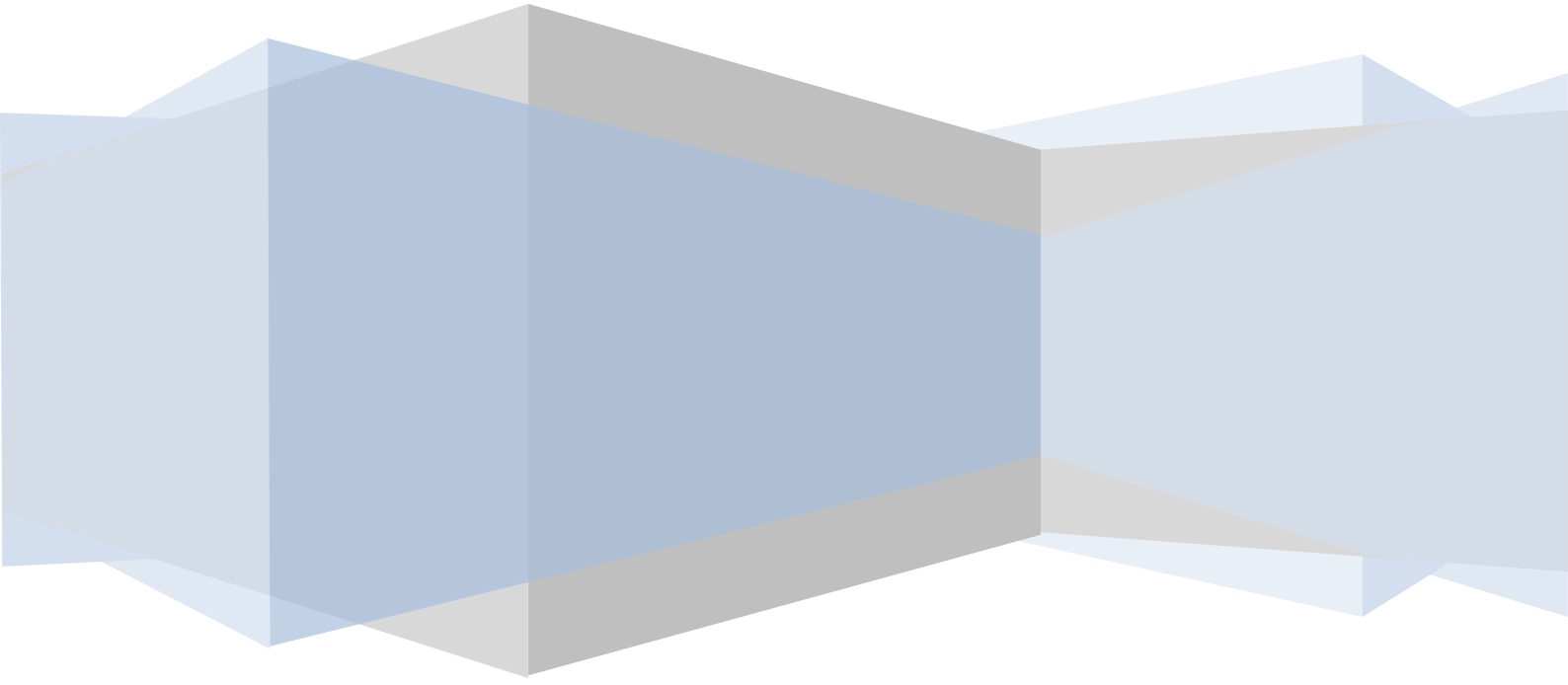
ROS - TF

An introduction to working with TFs

Bram v. d. Klundert



Inhoud

Notes

This document assumes you have knowledge of node to node communication and you have knowledge of the basics of ROS.

Tf

The package tf allows the robot to keep track of multiple coordinate frames and the relation between those frames. The frames are build in a tree structure, so one frame only has one parent but can have multiple children.

For example we have this robot:

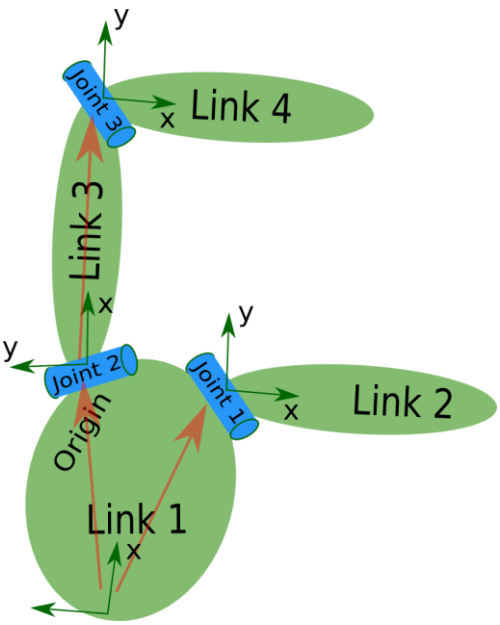


Figure1

As you can see there are a couple of frames in here. First of all we have the origin, this is the frame at the bottom of Link 1. Beside the origin we also have the frames of Link 2 and Link 3. These both are children of the origin. Finally we have the frame of Link 4 which is a child of Link 3.

In the next part of the example we will pretend our robot is 2d so it will only have an x axis and a y axis. This to make it a bit easier to understand.

Now say Link 3 would rotate so the y axis of its frame will point downwards.

Let's look at the result from a couple of different frames.

First the frame of Link 3. In the frame of Link 3 there won't be any changes. The coordinates and rotation of Link 4's frame will be exactly the same as before.

Now let's look at the rotation from the origin frame. The first thing that will have changed is the rotation of the frame of link 3. The x axis no longer points straight up. It now points to the left. Now if

we look at the location of Link 4's frame we will see that it has moved to the left and down. The frame has also rotated so now the x axis will point straight up.

Structure

all tf's have a tree structure, this means the structure start from an origin and then branches out in a way that every parent can have multiple children and a child can only have one parent.

In the example of figure1 Link1 has the origin frame. The frames of Link 3 and 2 are children of the origin and the frame of Link 4 is a child of the frame of Link 4.

How to use them

Before we are going to start with the code examples, we will need to make a package for it. Lets name this package: "learning_tf" and place it in the stack "ros_tutorials". For this tutorials the package needs the following dependencies: tf, roscpp and turtle_teleop.

Now we will take a look at how to use tf's and publish them. We will also look at how to transform from one frame to another and the problems that occur when doing so.

Tf broadcaster

First of all we are going to look at how to make a tf publisher. For this example we will use the turtlesim node to simulate it and show its effects. in this node we are going to listen to the topic pose which publishes the location of the turtle. After that we are going to publish a tf which shows the location of the turtle. Copy the following code and save it as: "turtle_tf_broadcaster.cpp"

The code

```
#include <ros/ros.h>
#include <tf/transform_broadcaster.h>
#include <turtlesim/Pose.h>

std::string turtle_name;



void poseCallback(const turtlesim::PoseConstPtr& msg){
  static tf::TransformBroadcaster br;
  tf::Transform transform;
  transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
  transform.setRotation( tf::Quaternion(msg->theta, 0, 0) );
  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
                   "world", turtle_name));
}

int main(int argc, char** argv){
  ros::init(argc, argv, "my_tf_broadcaster");
  if (argc != 2){ROS_ERROR("need turtle name as argument"); return -1;};
  turtle_name = argv[1];

  ros::NodeHandle node;
  ros::Subscriber sub = node.subscribe(turtle_name+"/pose", 10,
                                       &poseCallback);

  ros::spin();
  return 0;
};
```

Let's look at the most important parts.

```
#include <tf/transform_broadcaster.h>
#include <turtlesim/Pose.h>
```

We need the transform_broadcaster to broadcast any tfs and of course we need the pose to get the information we need from the turtlesim.

```
  static tf::TransformBroadcaster br;
```

Here we create a transform broadcaster

```
void poseCallback(const turtlesim::PoseConstPtr& msg){
  static tf::TransformBroadcaster br;
  tf::Transform transform;
  transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
  transform.setRotation( tf::Quaternion(msg->theta, 0, 0) );
  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
                   "world", turtle_name));
}
```

Here we create a transform to send, after that we fill it with the data we received from the turtlesim node. note that the rotation of the quaternion has the order z, y, x. so the theta is set to the rotation on the z axis.

```
  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(),
                   "world", turtle_name));
```

Let's take a closer look at the part where we send the transform. When we want to send a transform we need to give 4 parameters.

The first parameter we need to give is the transform we want to send.
The second parameter is a timestamp . Normally you will just stamp it with the current time, so we chose *ros::Time::now()* for it.
The third parameter is the name of the parent frame, in this case that's "world".
Finally we give the name of the frame we are going to publish in this example it's the name inside the variable turtle_name.

```
  if (argc != 2){ROS_ERROR("need turtle name as argument"); return -1;};
  turtle_name = argv[1];

  ros::NodeHandle node;
  ros::Subscriber sub = node.subscribe(turtle_name+"/pose", 10,
                                       &poseCallback);
```

Here we get the name of the turtle (from the argument we will give when we start the node). after that we make a subscriber to subscribe to the turtle and call poseCallback when a message is published.

```
  ros::spin();
  return 0;
};
```

Here we just spin the node.

Running the publisher

First we will need to build the publisher. To do this we will need to add the node to the CMakelist

```
  rosbuild_add_executable(turtle_tf_broadcaster src/turtle_tf_broadcaster.cpp)
```

Now we just have to build it by calling make or rosmake.

Now we are going to make a launch file as we want to launch more than one node:

```
<launch>
  <include file="$(find turtle_teleop)/launch/turtle_keyboard.launch" />

  <node pkg="learning_tf" type="turtle_tf_broadcaster"
        args="/turtle1" name="turtle1_tf_broadcaster" />
  <node pkg="learning_tf" type="turtle_tf_broadcaster"
        args="/turtle2" name="turtle2_tf_broadcaster" />

</launch>
```

Paste the above into a text file and save it as start_demo.launch.

The launch file will (when run) launch a couple of things. First of all it will run the turtle_keyboard launch file. Beside that it will also run two versions of the node we just wrote. One for each turtle we want to publish the tf for.

Now let's run the node by typing:

```
roslaunch learning_tf start_demo.launch
```

you should see a window with a turtle in it pop up. If you select the terminal you can drive around the turtle. To show what we just did open up a new terminal and run this command:

```
rosrun tf tf_echo /world /turtle1
```

if you drive the turtle around now you should see the values change. This means that it is publishing transforms and that they change according to the location of the turtle.

Tf listener

In this example we are going to write a listener for tf's that will look at the tf that is broadcasted by turtle 1 and make a second turtle chase the first one. Copy the code below and save it as: "turtle_tf_listener.cpp"

The code

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <turtlesim/Velocity.h>
#include <turtlesim/Spawn.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "my_tf_listener");

  ros::NodeHandle node;

  ros::service::waitForService("spawn");
  ros::ServiceClient add_turtle =
       node.serviceClient<turtlesim::Spawn>("spawn");
  turtlesim::Spawn srv;
  add_turtle.call(srv);

  ros::Publisher turtle_vel =
       node.advertise<turtlesim::Velocity>("turtle2/command_velocity", 10);

  tf::TransformListener listener;

  ros::Rate rate(10.0);
  while (node.ok()){
    tf::StampedTransform transform;
    try{
      listener.lookupTransform("/turtle2", "/turtle1",
                               ros::Time(0), transform);
    }
    catch (tf::TransformException ex){
      ROS_ERROR("%s",ex.what());
    }

    turtlesim::Velocity vel_msg;
    vel_msg.angular = 4 * atan2(transform.getOrigin().y(),
                                transform.getOrigin().x());
    vel_msg.linear = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                pow(transform.getOrigin().y(), 2));
    turtle_vel.publish(vel_msg);

    rate.sleep();
  }
  return 0;
};
```

Let's take a look at the most important parts.

```
#include <tf/transform_listener.h>
#include <turtlesim/Velocity.h>
#include <turtlesim/Spawn.h>
```

Not much special here, we just include the messages and service we are going to use. Beside that we include transform_listener as we want to listen to transforms.

```
ros::service::waitForService("spawn");
ros::ServiceClient add_turtle =
     node.serviceClient<turtlesim::Spawn>("spawn");
turtlesim::Spawn srv;
add_turtle.call(srv);
```

Here we spawn a second turtle to follow the first. We do this by calling a service of turtlesim.

```
ros::Publisher turtle_vel =
     node.advertise<turtlesim::Velocity>("turtle2/command_velocity", 10);

tf::TransformListener listener;
```

Here we just create a publisher so we can let the second turtle move. we also create a transform listener.

```
    try{
      listener.lookupTransform("/turtle2", "/turtle1",
                               ros::Time(0), transform);
    }
    catch (tf::TransformException ex){
      ROS_ERROR("%s",ex.what());
    }
```

Here we look up the transform between turtle2 and turtle1. We do that for the last available (ros::Time(0)) and store it in the transform we had just made. We also catch tf exceptions so our program doesn't crash when something goes wrong while looking up the transform.

```
    turtlesim::Velocity vel_msg;
    vel_msg.angular = 4 * atan2(transform.getOrigin().y(),
                                transform.getOrigin().x());
    vel_msg.linear = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                pow(transform.getOrigin().y(), 2));
    turtle_vel.publish(vel_msg);
```

Now we just have to publish the movement the second turtle has to make to follow the first turtle.

Running your node

First we will need to build the publisher. To do this we will need to add the node to the CMakelist

```
rosbuild_add_executable(turtle_tf_listener src/turtle_tf_listener.cpp)
```

Now we just have to build it by calling make or rosmake.

Now let's edit the launch file of the previous example to also start this node. to do this add the following line to the start_demo.launch file.

```
<node pkg="learning_tf" type="turtle_tf_listener" name="listener" />
```

Now let's run the node by typing:

```
roslaunch learning_tf start_demo.launch
```

You should see a window again this time it should see two two turtle, where one is going to chase after the other. You might see two of these error messages:

```
[ERROR] 1253915565.300572000: Frame id /turtle2 does not exist! When trying to
transform between /turtle1 and /turtle2.
[ERROR] 1253915565.401172000: Frame id /turtle2 does not exist! When trying to
transform between /turtle1 and /turtle2.
```

These messages show because we try to transform between the frame of turtle1 and the frame of turtle2 before turtle2 has spawned.

Tf and time

The tf package allows you to look up transform up to 10 seconds in the past. In the next example we are going to use this to chase the turtle where it was 5 seconds in the past. To do this we are going to change the turtle_tf_listener we made in the previous example.

The code

```
#include <ros/ros.h>
#include <tf/transform_listener.h>
#include <turtlesim/Velocity.h>
#include <turtlesim/Spawn.h>

int main(int argc, char** argv){
  ros::init(argc, argv, "my_tf_listener");

  ros::NodeHandle node;

  ros::service::waitForService("spawn");
  ros::ServiceClient add_turtle =
       node.serviceClient<turtlesim::Spawn>("spawn");
  turtlesim::Spawn srv;
  add_turtle.call(srv);

  ros::Publisher turtle_vel =
       node.advertise<turtlesim::Velocity>("turtle2/command_velocity", 10);

  tf::TransformListener listener;

  ros::Rate rate(10.0);
  while (node.ok()){
    tf::StampedTransform transform;
    try{
      ros::Time now = ros::Time::now();
      ros::Time past = now - ros::Duration(5.0);
      listener.waitForTransform("/turtle2", now,
                                "/turtle1", past,
                                "/world", ros::Duration(1.0));
      listener.lookupTransform("/turtle2", now,
                               "/turtle1", past,
                               "/world", transform);
    catch (tf::TransformException ex){
      ROS_ERROR("%s",ex.what());
    }

    turtlesim::Velocity vel_msg;
    vel_msg.angular = 4 * atan2(transform.getOrigin().y(),
                                transform.getOrigin().x());
    vel_msg.linear = 0.5 * sqrt(pow(transform.getOrigin().x(), 2) +
                                pow(transform.getOrigin().y(), 2));
    turtle_vel.publish(vel_msg);

    rate.sleep();
  }
  return 0;
};
```

Now let's take a look at the change:

```
try{
  ros::Time now = ros::Time::now();
  ros::Time past = now - ros::Duration(5.0);
  listener.waitForTransform("/turtle2", now,
                            "/turtle1", past,
                            "/world", ros::Duration(1.0));
  listener.lookupTransform("/turtle2", now,
                           "/turtle1", past,
                           "/world", transform);
```

A couple of things changed here

First of all we start by making 2 variable for time, one (now) for the current time and a second (past)for the time 5 seconds in the past.

Now we also wait for a transform to become available. we are waiting for a transform to become available because else the program will complain there isn't an transform available at the time we request it(now) this is a couple of milliseconds difference. Also note that we give two times when we look up the transform. We say: give me the transform from this frame(turtle2) at this time(now) to this frame(turtle1) at this time(past) with this fixed frame(world) and save it in this variable(transform). The wait for transform is nearly the same except that the last parameter is the timeout (1 second here)

Running your node

Just call make or rosmake (as the node is already in the CMakelist)

Urdf(unified Robot Description File)

the urdf file is an XML specification to describe a robot. Urdf files are used for: a kinematic and dynamic description of the robot, a visual representation of the robot and a collision model of the robot. Beside that you can also use urdf files to simulate your robot if you add some gazebo tags.

First we will look at some of the basics tags, after that we will look at a way of making your urdf more readable finally we are going to look what tags we need to add to show the robot in the gazebo simulation.

Syntax

First we are going to look at a small example of an urdf file of the example in figure 1.

```
<robot name="test_robot">
  <link name="link1" />
  <link name="link2" />
  <link name="link3" />
  <link name="link4" />

  <joint name="joint1" type="continuous">
    <parent link="link1"/>
    <child link="link2"/>
    <origin xyz="5 3 0" rpy="0 0 0" />
    <axis xyz="-0.9 0.15 0" />
  </joint>

  <joint name="joint2" type="continuous">
    <parent link="link1"/>
    <child link="link3"/>
    <origin xyz="-2 5 0" rpy="0 0 1.57" />
    <axis xyz="-0.707 0.707 0" />
  </joint>

  <joint name="joint3" type="continuous">
    <parent link="link3"/>
    <child link="link4"/>
    <origin xyz="5 0 0" rpy="0 0 -1.57" />
    <axis xyz="0.707 -0.707 0" />
  </joint>
</robot>
```

Now let's discuss the tags that were used in there. There is one more tag(the <gazebo> tag but that one will be explained in the chapter gazebo.

The robot element

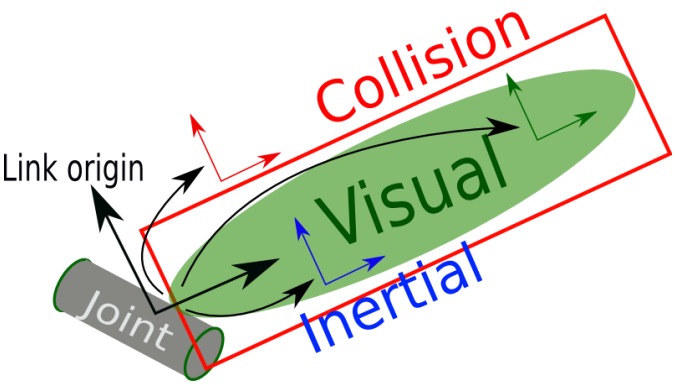
```
<robot name="test_robot">
```

The first tag we see is the <robot> tag. In this tag you give your robot a name. it also tells the program that will convert the file where the start and end is. Like all tags in XML you will need to close it with </robot> at the end of the document.

The link element

```
  <link name="link1" />
  <link name="link2" />
  <link name="link3" />
  <link name="link4" />
```

Links define the solid parts of your robot. Every link needs a name. but elements can also have other elements: inertial, visual and collision.



Inertial, visual and collision are added to the link in a slightly different way than the name is. To add them we have to open with a link tag than add the things we want to add and eventually close it again. An example of this is shown below:

```
<link name="my_link">
   <inertial>
     <origin xyz="0 0 0.5" rpy="0 0 0"/>
     <mass value="1"/>
     <inertia ixx="100"  ixy="0"  ixz="0" iyy="100"
iyz="0" izz="100" />
   </inertial>

   <visual>
     <origin xyz="0 0 0" rpy="0 0 0" />
     <geometry>
       <box size="1 1 1" />
     </geometry>
     <material name="Cyan">
       <color rgba="0 255 255 1.0"/>
     </material>
   </visual>

   <collision>
     <origin xyz="0 0 0" rpy="0 0 0"/>
     <geometry>
       <cylinder radius="1" length="0.5"/>
     </geometry>
     <contact_coefficients mu="0" kp="1000.0" kd="1.0"/>
   </collision>
 </link>
```

The inertial element is optional and can be used to add any inertial properties to the link. Within this tag you have to option to add the following elements: origin, mass and inertia.

Origin is an optional tag that allows you to change the origin of the inertial.
Mass is a required tag that allows you to specify the weight of the link.
Inertia is a required tag that allows you to specify the inertia matrix for the link. Because the matrix is symmetric you only have to specify the above diagonal elements. This is done with the ixx, ixy, ixz, iyy, iyz and izz attributes.

The visual element is optional and can be used to specify the shape of the object for visualization in rviz. Within this tag you have the option to add the following elements: origin, geometry and material.

Origin is the same as described at Inertial.
Geometry is a required tag that allows you set the shape of the object to an: box, cylinder or sphere. You can also use it to load a custom mesh for your part.
Material is an optional tag that allows you to set a color or texture for the link. The material tag needs a name for the color/texture.

the collision element is optional and can be used to specify the collision properties of the link. Within this tag you have the option to add the following elements: origin, geometry and contact_coefficients.

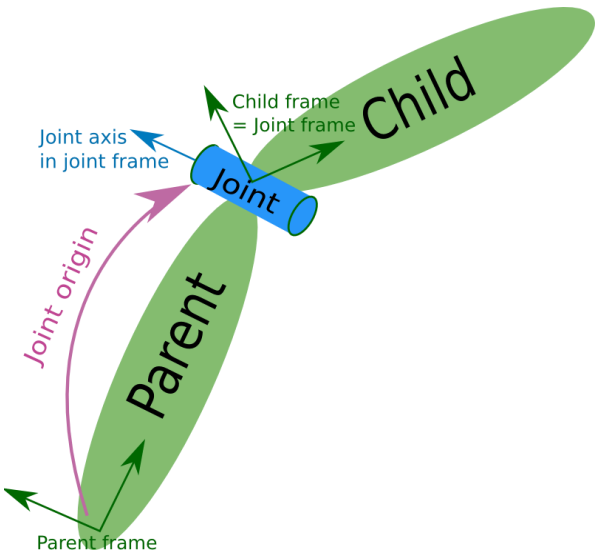
Origin is the same as described at Inertial.
Geometry is the same as described at visual.
Contact_coefficients allows you to specify the interaction between two links when they collide. This interaction is defined by mu, kp and kd

The joint element

```
<joint name="joint1" type="continuous">
  <parent link="link1"/>
  <child link="link2"/>
  <origin xyz="5 3 0" rpy="0 0 0" />
  <axis xyz="-0.9 0.15 0" />
</joint>
```

The joint element is used to define the joints that link the different links together. It also defines safety limits of the joint. The joint tag needs two attributes, a name and a joint type. The available joint types are:

- revolute - a hinge joint that rotates along the axis and has a limited range specified by the upper and lower limits.
- continuous - a continuous hinge joint that rotates around the axis and has not upper and lower limits
- prismatic - a sliding joint that slides along the axis, and has a limited range specified by the upper and lower limits.
- fixed - This is not really a joint because it cannot move. All degrees of freedom are locked. This type of joint does not require the axis, calibration, dynamics, limits or safety_controller.
- floating - This is not really a joint because all 6 degrees of freedom are free.
- planar - This joint allows motion in a plane perpendicular to the axis.



The available elements are: origin, parent, child, calibration, axis, dynamics, limit and safety_controller
Just like at the link element those are added by placing them within the joint tags.

```
<joint name="my_joint" type="floating">
    <origin xyz="0 0 1" rpy="0 0 3.1416"/>
    <parent link="link1"/>
    <child link="link2"/>
    <calibration upper="0.0"/>
    <dynamics damping="0.0" friction="0.0"/>
    <limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
    <safety_controller k_velocity="10"/>
 </joint>
```

The origin element is a optional element that allows you to specify the location of the joint in the frame of the parent. This will also specify the origin of the child link.

The parent element is a required element that specifies the parent link of the joint. It has the required attribute “link” which specifies the name of the parent link.

The child element is a required element which specifies the child link of the joint. It has the required attribute “link” which specifies the name of the child link.

The axis element is an optional element that allows you to specify the directions in which the joint can move. This element isn’t used by fixed and floating joints.

the calibration element is an optional element which allows you to set a reference position for the joint, which is used to calibrate the absolute position of the joint.

The dynamics element is an optional element that can be used to specify physical properties of the joint, which are mainly used in the simulation. It has an attribute for damping and for friction.

The limit element is a tag which is required for revolute and prismatic joints. It is used to set the maximum effort and the maximum velocity of a joint. It can also be used to set a upper and lower limit on the movement of the joint.

The safety_controller element is an optional element used to reduce the speed near the upper and lower limits preventing the robot from bumping into them suddenly. The attributes soft_lower_limit and soft_upper_limit are optional and set the points from where the movement will be dampened. The k_position is optional and specifies the relation between position and velocity limits. The k_velocity is required and specify the relation between effort and velocity limits.

Validating your URDF file

Because making a good URDE file takes a lot of work, there is a large chance of small mistakes being made. Luckily the URDF package includes some tools to check your URDF file.

The first tool is check_urdf. You can call check_urdf with the following command:

```
Rosrun urdf check_urdf <your urdf file>
```

Where you have to replace <your urdf file> with the URDF file you made.

Check_urdf checks your urdf file and if no problem are in it, it will print a tree structure showing all links. If something is wrong in the fill it will print an error message which points to the part that's wrong.

The second tool is urdf_to_graphiz. It is called with the following command:

```
Rosrun urdf urdf_to_graphiz <your urdf file>
```

Where you have to replace <your urdf file> with the URDF file you made.

Urdf_to_graphiz generates a pdf file which shows a diagram with the connections between links and joints, giving a easily readable tree structure.

Xacro

Xacro is a macro language for xml, allowing you to create macros and thus making your xml files more readable. Especially when you have a large file.

Im going to explain a couple of functions of xacro. first of all we will look at a simple macro block. After that we discuss property's and property blocks. Third up is how to use some maths within xacro and finally we will look how we can use rospack commands in xacro.

Macro

Before we take a look at the options of the macro we will take a look at a small example that will show the basics of the xacro:macro.

```
<xacro:macro name="pr2_arm" params="suffix parent reflect">
  <pr2_upperarm suffix="${suffix}" reflect="${reflect}" parent="${parent}" />
  <pr2_forearm suffix="${suffix}" reflect="${reflect}"
parent="elbow_flex_${suffix}" />
</xacro:macro>

<xacro:pr2_arm suffix="left" reflect="1" parent="torso" />
<xacro:pr2_arm suffix="right" reflect="-1" parent="torso" />
```

That example would expand to:

```
<pr2_upperarm suffix="left" reflect="1" parent="torso" />
<pr2_forearm suffix="left" reflect="1" parent="elbow_flex_left" />
<pr2_upperarm suffix="right" reflect="-1" parent="torso" />
<pr2_forearm suffix="right" reflect="-1" parent="elbow_flex_right" />
```

Note that this makes it easy to define parts. If you would create definitions for pr2_upperarm and pr2_forearm you could create an entire arm with all its properties by writing a single line.

Now let's take a closer look at the sintax. There are a number of things that can be seen from this first line:

```
<xacro:macro name="pr2_arm" params="suffix parent reflect">
```

The first thing you want to note is the name parameter. The name is used when you want to call the macro later the name of the macro will then be: xacro: "name". after that there is the params parameter. The params parameter defines what parameters have to be added when you call your macro. In this case there will be 3 parameter given suffix, parent and reflect. If you place an '*' before the name of the parameter it means that it's a block property. You will have to define it like this inside the macro:

```
<xacro:insert_block name="*name*" />
```

Where you have to replace *name* with the name of the property (without the '*').

You can use macros inside macros, if this happens they are expended so that the macro on the outside gets expanded first.

Property and property blocks

Property blocks are blocks that allow you to define and use property on multiple places in a file.

```
<xacro:property name="the_radius" value="2.1" />
<xacro:property name="the_length" value="4.5" />

<geometry type="cylinder" radius="${the_radius}" length="${the_length}" />
```

The cylinder in the previous example would have a radius of 2.1 and a length of 4.5. Note how the properties are used by enclosing the name in $(). Properties especially become useful if you want to

use a value at multiple places, so you only have to change one value instead the value all over the place.

You can also create property block:

```
<xacro:property name="front_left_origin">
  <origin xyz="0.3 0 0" rpy="0 0 0" />
</xacro:property>

<pr2_wheel name="front_left_wheel">
  <xacro:insert_block name="front_left_origin" />
</pr2_wheel>
```

This way you only have to chance one thing to change the origin of all the parts.

Math expressions

Math expressions allow you to do basic arithmetics and use variables. To do any maths in your urdf file include your calculation in ${}. Example:

```
<xacro:property name="pi" value="3.1415926535897931" />

<circle circumference="${2.5 * pi}" />
```

Gazebo

The gazebo package contains a resent version of Gazebo and some ros specific patches to allow it to work with ros.

What is gazebo?

Gazebo is a third party multi-robot simulation for outdoor environments. It works in 3d and allows you to simulate everything from sensors to actuators and other objects (including collision ect). Like ros it also is an open source project.

Pros and cons of gazebo

In this section i would like to explain some reasons why or why not to use gazebo with your robot.

Pros:

- gazebo allows you to simulate your robot. Including collisions and any sensor data you would get in the real world. Of course this is great as it allows you to test your robot in an environment where you can't damage anything.
- Gazebo allows you to simulate with the controllers you would also use while controlling your real robot.
- You will be able to get actual sensor data out of your simulation

Cons

- gazebo can really be slow, especially when you start to do more complicated environments and/or robots. An example is that it takes more than a half hour for the pr2 to pick up a coffee cup in simulation.
- Gazebo can be a bit difficult to get working right, and also to get used to the environment.
- You will have to make a full URDF model of your robot. This can be a lot of work to make and you will need a lot of info on the robot.

the gazebo tag

the gazebo tag is only used to make the simulation more realistic by adding friction contact stiffness and contact damping to the model. It also has an option to enable self collide for the robot, which will allow the robot to collide with anything in the world.

We will start with an example than explain all the parts of the example.

```
<gazebo reference="my_box">
    <mu1 value=”100.0” />
    <mu2 value=”100.0” />
    <kp  value=”1000000.0” />
    <kd  value=”1.0” />
    <material>Gazebo/Blue</material>
</gazebo>
```

Reference: this is the link where the gazebo is applying to.

MU1 and MU2: with these two elements you can define the friction coefficients for the two principle contact directions.

KP: with this element you can define the contact stiffness.

KD: with this element you can define the contact damping

Material: this allows you to set the material/color of the link.