Robot Modelling and Control in ROS

Laurent Lequièvre
Juan Antonio Corrales Ramon
Institut Pascal – Clermont-Ferrand - France

ROBOT MODELLING WITH URDF

Index

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Robot description package

- In ROS, we define a package for each robot that we want to model:
  - **Kinematic model**: Static transformations between links/joints
  - **Dynamic model**: Mass and inertia of links
  - **Visual representation**: Detailed 3D representation of links
  - **Collision model**: Simplified 3D representation for collision checking

- Normally this **robot description package** contains the next folders:
  - **/meshes**: CAO files with 3D models (STL or DAE) of links
  - **/urdf**: Models of the robot in URDF/xacro format
  - **/launch**: Scripts for accessing and visualizing the robot model

- Create new package with these sub-folders and copy provided mesh files:
  catkin_create_pkg gripper_description roscpp tf geometry_msgs urdf rviz xacro

ROS packages in robot modelling

- The ROS meta-package **robot_model** contains packages needed in robot modelling:
  - **urdf**: An XML robot description format and parser
  - **kdl_parser**: A parser to create kinematic and dynamic models from urdf
  - **robot_state_publisher**: A publisher of tf for the 3D pose of each link
  - **resource_retriever:** Loader of url-format data files into memory
  - **collada_urdf, collada_parser...**: Transformation tools for other formats

- Additional packages for working with robot models:
  - **Rviz**: 3D visualization tool for ROS that can load URDF files
  - **xacro**: XML macros language for getting shorter and readable XML files

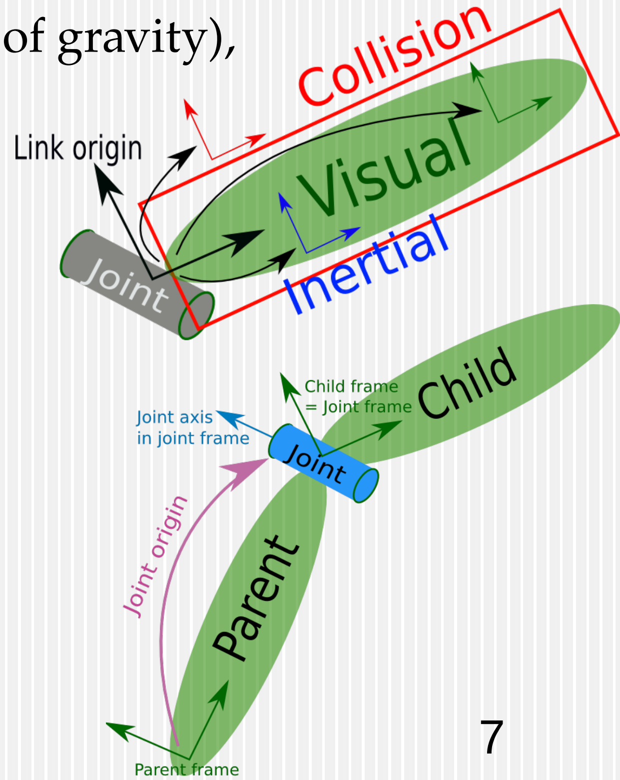
Introduction to URDF

- **URDF** (Unified Robot Description Format) is a XML format for representing robot models and sensors. It covers:
  - Kinematic and dynamic description of the robot
  - Visual representation of the robot
  - Collision model of the robot
- The **urdf** package contains a C++ parser for reading files in URDF format and tools for verifying and visualizing these files.
- URDF presents several limitations:
  - Only **one robot** per file. Multiple robots require the use of xacro.
  - Only **tree structures** can be used. Parallel robots cannot be handled.
  - Only **rigid links** can be used. Flexible elements are not possible.
  - Future improvements: URDF 2 or other formats (SDF- Gazebo-…).

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

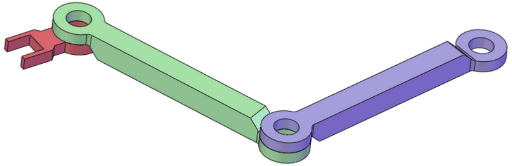
XML Tags in URDF

- The description of a robot consists of a set of links connected by joints:
  - **<robot>**: Root tag of the entire robot
  - **<link>**: Definition of a link with inertial (centre of gravity), visual and collision frames
    - **<inertial>:** origin, mass, inertia
    - **<visual>:** origin, geometry, material
    - **<collision>:** origin, geometry (shape/mesh)
  - **<joint>**: Definition of a joint between two links with different types (revolute, prismatic,fixed)
    - **<parent link=”link1”/>**
    - **<child link=”link2”/>**
    - **<origin>** (child frame wrt parent) and **<axis>**



INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

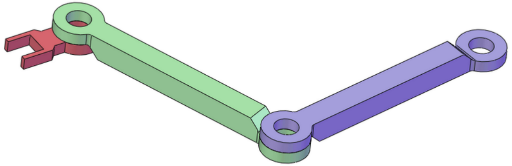
First URDF model



- Example of a 3-joint planar robot with links of 0.5m.
  - Insert initial/end tags: <robot name="planar_3dof"> </robot>
  - Add a “virtual link” to represent the kinematic base frame of the robot: <link name="base_link"/>
  - Add the first arm link:
    - Create the link tag: <link name="link_1"> </link>
    - Add inside the tag the visual data of the link (mesh and material):

```
<visual>
<geometry>
<mesh filename="package://gripper_description/meshes/visual/arm_link.stl"/>
<material name="grey"> <color rgba="0.7 0.7 0.7 1.0"/> </material>
</geometry>
</visual>
```

First URDF model



- Example of a 3-joint planar robot with links of 0.5m.
  - Add inside the tag the collision data of the link (mesh):

```
<collision><geometry>
<mesh filename="package://gripper_description/meshes/collision/arm_link.stl"/
>
</geometry></collision>
```

- Add the information of the joint (parent, child, origin, axis, limits):

```
<joint name="joint_1" type="revolute">
<parent link="base_link"> <child link="link_1"/>
<origin xyz="0 0 0" rpy="0 0 0" /> <axis xyz="0 0 1" />
<limit lower="-1.57" upper="1.57" effort="0" velocity="0.5" /></joint>
```

- Add two links (*link_2* and *gripper*) and two joints (*joint_2* and *joint_3*).

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

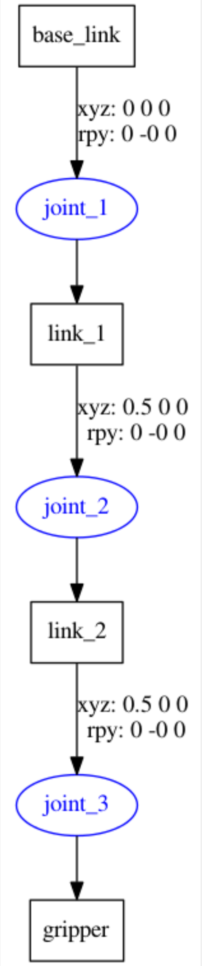
Testing URDF with commands

- Testing the elements of the URDF: check_urdf planar_3dof.urdf

  robot name is: planar_3dof
  ---------- Successfully Parsed XML ---------------
  root Link: base_link has 1 child(ren)
      child(1): link_1
          child(1): link_2
              child(1): gripper

  If the command is not available: sudo apt-get install liburdfdom-tools

- Visualizing URDF in pdf: urdf_to_graphiz planar_3dof.urdf
  To view generated pdf file: evince planar_3dof.pdf



Testing URDF in Rviz

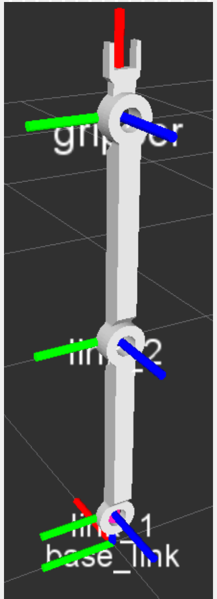
- Create a script (file “display.launch”) for visualizing URDF file in Rviz:

```
<launch>
    <param name=”robot_description” textfile="$(find gripper_description)/urdf/planar_3dof.urdf"/>
    <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 gripper_description)/urdf/urdf.rviz"
    required=”true”/>
</launch>
```

  If the package urdf_tutorial is not available: sudo apt-get install ros-indigo-urdf-tutorial

- This script (stored in the “launch” sub-folder) does 3 steps:
  - **Loads the URDF into the parameter “robot_description”**
  - **Runs nodes to publish the robot state** (robot_state/joint_state)
  - **Starts Rviz** with a predefined config file and reads robot_description
    - Firstly, rviz config file (urdf.rviz) is not available, create it and store it to urdf folder:

      1. add **“RobotModel”** element in left tree of Rviz
      2. add **“TF”** element in left tree of Rviz
      3. define **Fixed Frame=”base_link”** in “Global Options”



INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

robot_description as parameter

- By convention, the URDF file of a robot should be stored as a the **parameter "robot_description"** in the parameter server for later use.
- The **parameter server** is a shared, multi-variable dictionary (pairs "name-value") stored inside the ROS master and accessible by ROS nodes. Since it is not optimized, it is used for static data (**configuration parameters**).
  - **List all parameters:** rosparam list
  - **Get one parameter value:** rosparam get /robot_description
  - **Delete a parameter:** rosparam delete /robot_description
  - **Set one parameter value (single, list, file, dictionary-as a namespace-):**
    rosparam set /color "[150,55,210]" #List;
    rosparam set /robot_description -t planar_3dof.urdf # Contents of a file
    rosparam set /gain/p 10 ; rosparam set /gain/i 20; rosparam set /gain/d 30;
  - **Store/load all parameters to YAML file:** rosparam dump/load parameters.yaml

Introduction to xacro

- The flexibility of URDF reduces with **complex robot models**.
- **Xacro** (XML Macros) is an XML macro language that improves URDF by adding:
  - **Simplicity:** Xacro defines macros inside the robot description and reuses them. Thereby, the code is shorter, more readable and simpler.
  - **Modularity and reusability**: It can include macros from other files so that the robot model can be organized in blocks that can be reused where necessary.
  - **Programmability:** xacro supports simple programming elements such as variables, conditional statements, constants and mathematical expressions.
- A xacro file will be read by the **xacro program** that will run all its macros and output the result (normally to a final urdf file):

  rosrun xacro xacro.py model.xacro > model.urdf

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

XML Tags in xacro (I)

- **<xacro:include>**: Import the content from another file.

  ```
  <xacro:include filename="$(find gripper_description)/urdf/planar_3dof.urdf.xacro"/>
  ```

- **<xacro:property>:** Definition of constant values for later use.
  - **Definition of the property**:

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

  - **Use of the property** with ${property_name}, including math operations (+,-,*,/):

  ```
  <limit lower=”${-pi/2.0}” upper=”${pi/2.0}” effort=”0” velocity=”0.5” />
  ```

- **<xacro:macro>:** Macro with parameters whose body will be replaced when used.
  - **Definition of the macro**:

  ```
  <xacro:macro name="default_inertial" params="mass">
    <inertial> <mass value=”${mass}”/>
    <inertia ixx=”1.0” ixy=”0.0” ixz=”0.0” iyy=”1.0” iyz=”0.0” izz=”1.0” /> </inertial>
  </xacro>
  ```

XML Tags in xacro (II)

- **<xacro:macro>:**
  - **Use of the macro** by calling it with its name and filling the required parameters:

```
<xacro:default_inertial mass=”10”>
```

- **<xacro:macro>:** Even entire blocks can be used as parameters for macros.
  - **Definition:** mark block parameter with * and insert it with <xacro:insert_block>:

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

  - **Use:** Expand the xacro by defining normal parameters and block parameters values:

```
<xacro:link_shape name="base_link">
   <cylinder radius=”0.42” length=”0.01”/>
</xacro:link_shape>
```

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

URDF simplification with xacro

- Create a new xacro file (planar_3dof.xacro) in the urdf folder that includes:
  - Definition of xacro properties for: pi, link_length(0.5), base_height(0.1) and vel_max(0.5)
  - Definition of xacro macro for link definition with 3 parameters: link_name, visual_mesh and collision_mesh

- Create a new launch file (display_xacro.launch) for this xacro by modifying the previous launch. Use the xacro.py program in order to translate xacro into urdf:

```
<launch>
        <param name="robot_description" command="$(find xacro)/xacro.py '$(find gripper_description)/urdf/planar_3dof.xacro'" />
        <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 gripper_description)/urdf/urdf.rviz" required="true" />
    </launch>
```

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Index

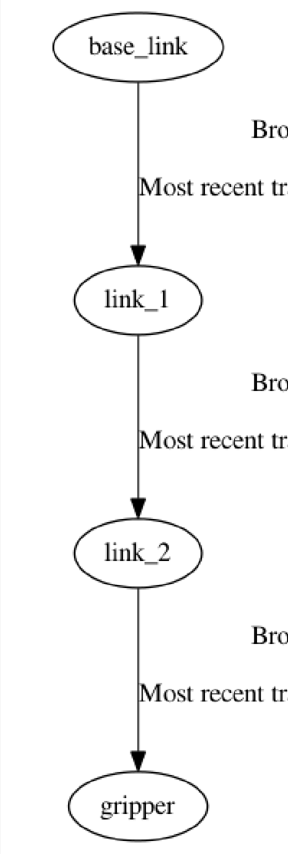
- Robot Modelling with URDF
  - Robot description package
  - First URDF model
  - Rviz
  - Modelling with xacro
- Determining Robot State
  - Joint State Publisher
  - Robot State Publisher
  - tf

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Determining robot state

Following the robot state in ROS

- **joint_state_publisher:**
  - This package publishes **sensor_msgs/JointState** messages for a robot.
  - This package reads the robot_description parameter, finds all non-fixed joints and **publishes a JointState message with all those joints values**.
  - For controlling JointState with GUI sliders in simulation, define the parameter use_gui as true by adding this line in the launch file: <param name="use_gui" value="true" />
  - Set manually param if GUI is missing: **rosparam set /use_gui true**
    - If error: sudo apt-get update; sudo apt-get install ros-melodic-joint-state-publisher-gui
  - Verify joint_state with topic: rostopic echo /joinstates
- **robot_state_publisher:**
  - It uses the URDF from robot_description parameter and the joint positions from the topic joint_states to **calculate forward kinematics and publish it via tf**.
  - Tree of tf: rosrun tf view_frames
  - tf between two frames: rosrun tf tf_echo base_link gripper



INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

ROBOT CONTROL WITH ROS_CONTROL

Index

- ROS controllers
  - Architecture of ros_control
  - Controller manager
  - Sending commands

Introduction to ros_control

- **ros_control** packages are a rewrite of pr2_mechanism package to make generic controllers for all robots:
  - Inputs: Joint state data of the robot (encoders) + Set point (goal).
  - Outputs: Joint commands (Effort/Angle) for driving robot to goal.
  - Basis: Control loop feedback (PID controllers) to generate output.

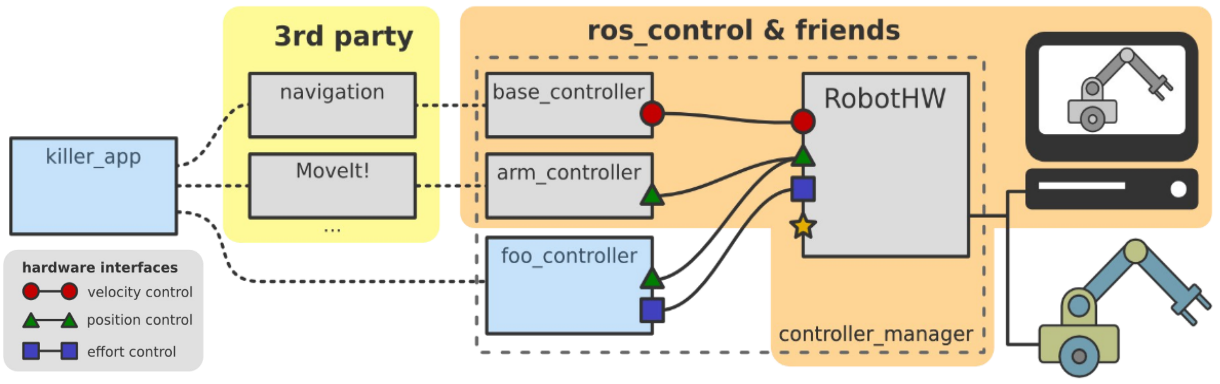
ros_control

- **Packages** inside ros_control:
  - **control_toolbox:** Common modules (PID and Sine) for controllers.
  - **controller_interface:** Interface base class for controllers.
  - **controller_manager:** Manager to load/unload/start/stop controllers.
  - **controller_manager_msgs:** Message and service definitions for controller manager.
  - **hardware_interface:** Base class for hardware interfaces.
  - **transmission_interface:** Interface classes for the transmission interface.

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Architecture of ros_control

- Goals:
  - Reuse control code
  - Abstraction of HW for ROS
  - Ready-to-use tools
  - Common controllers for real and simulation



- Sequence of events in ros_control:
  - **Planning tools** ('navigation' in mobile and 'MoveIt!' in manipulators): Establish the goals (set points) for the controllers according to environment constraints.
  - **ROS controllers**: Feedback mechanism (PID loop) which receives a set point and control the output (position, effort or velocity) using the feedback from actuators.
  - **Hardware interfaces:** Mediator between ROS controllers and the real hardware or simulator. It is a software representation of the robot and abstraction of hardware.

ROS controllers

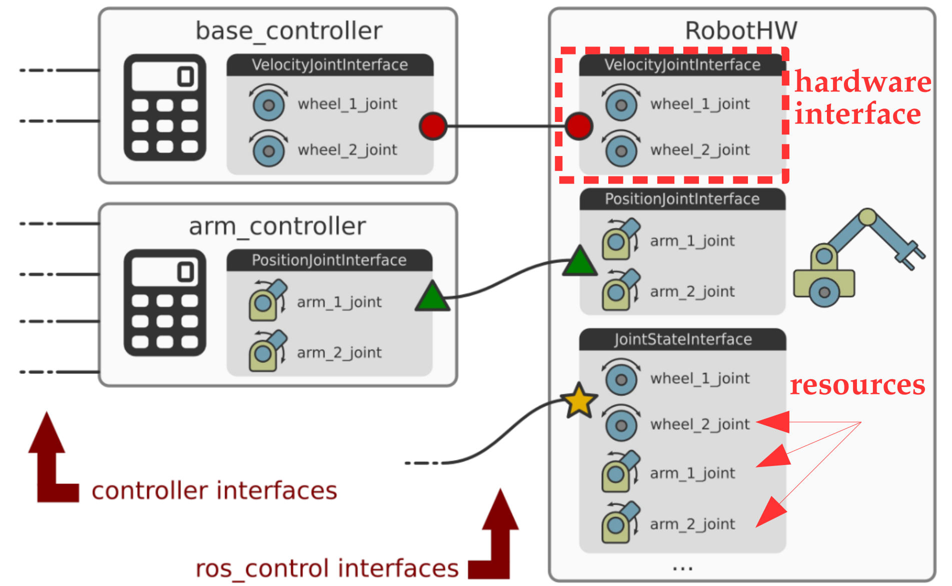
- Sensor state reporting:
  - **joint_state_controller**: Publishes sensor_msgs/JointState topics
  - **imu_sensor_controller**: Publishes sensor_msgs/Imu topics
  - **force_torque_sensor_controller**: Publishes geometry_msgs/Wrench topics

- Actuators and joints controllers in different control spaces:
  - **Effort controllers** (fixing torques for joints): *joint_effort_controller, joint_group_effort_controller, joint_position_controller, joint_velocity_controller*
  - **Position controllers** (fixing angles for joints): *joint_position_controller, joint_group_position_controller*
  - **Velocity controllers** (fixing angular velocities for joints): *joint_velocity_controller, joint_group_velocity_controller, joint_position_controller*
  - **Trajectory controllers** (fixing joint-space trajectories on a group of joints).
  - **diff_driver_controller** (differential drive wheel system with twist commands).

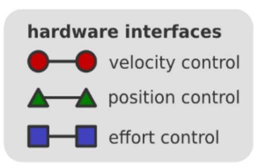
INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Hardware Interfaces

- Abstraction of robot hardware:
  - **Resource**: actuators, joints, sensors
  - **Interface**: set of similar resources
  - **Robot**: set of interfaces
- **Allocation of resources** for controllers, with corresponding hardware interfaces:
  - Read-only (Get states of resources): *joint/actuator state,* *IMU sensor,* *force-torque sensor*
  - Read-write (Send commands to resources): *position joint/actuator,* *velocity joint/actuator,* *effort joint/actuator,*

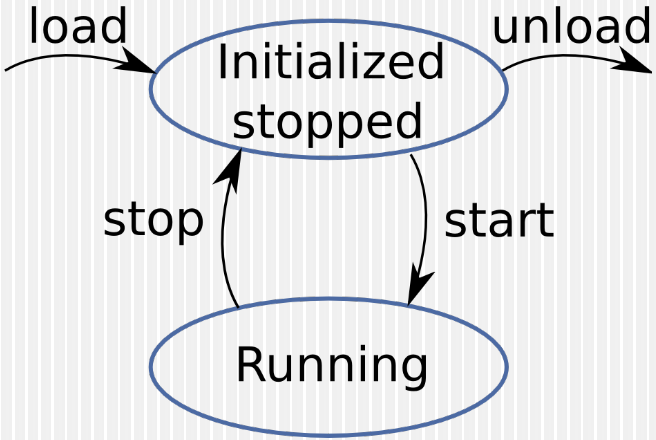


Communication between controllers and hardware interfaces



INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

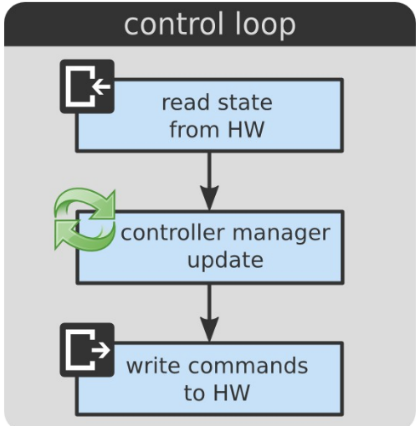
Controller manager



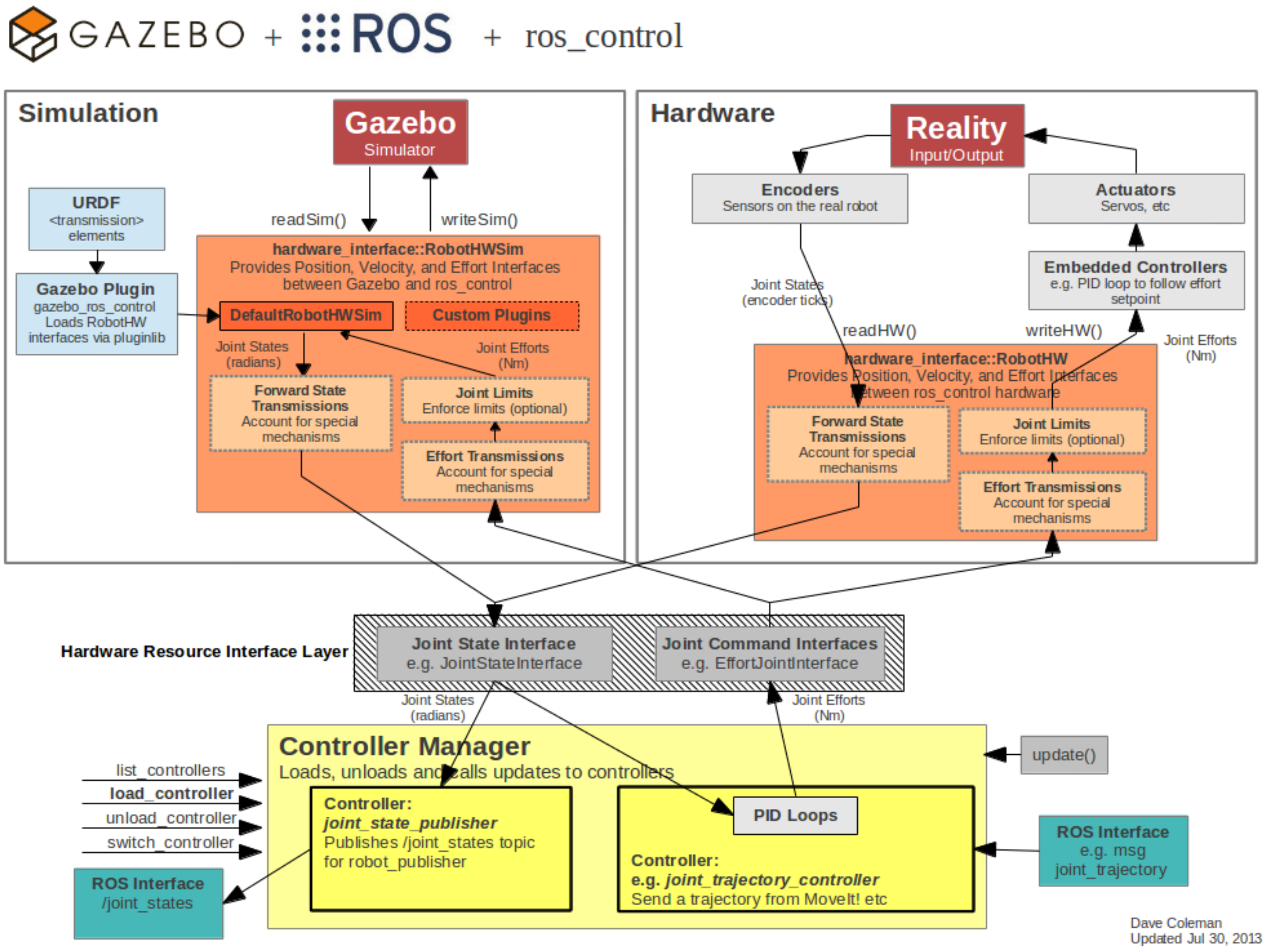
- It provides the infrastructure to interact with controllers (as plugins) and change their states:
  - **load**: load a controller (construct and initialize)
  - **unload**: unload a controller (destroy)
  - **start**: start a controller
  - **stop:** stop a controller
  - **spawn:** load and start a controller
  - **kill:** stop and unload a controller

  rosrun controller_manager controller_manager <command> <controller_name>
- The hardware interfaces and resources are accessible to the controller manager (cm) through a **RobotHW class instance** (robot):

  In the control loop, at each step:
  1. Read RobotHW state: *robot.read()*
  2. Controller manager updates all running controllers: *cm.update()*
  3. Write commands to RobotHW: *robot.write()*



Gazebo+ ros_control



INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

URDF extension for robot simulation in Gazebo

- In order to simulate in Gazebo, the URDF-xacro has to be completed with :
  - **<inertial>**: The dynamic model of each link (origin/mass/inertia)
  - **<gazebo>** with optional settings for links/joints (moved to rrbot.gazebo):
    - **<material>:** gazebo material (standard URDF materials for Rviz are not applicable)
    - **<mu1/mu2>**: friction coefficients for contact simulation with ODE ...(See gazebo doc for more).
  - Add a “world” link with a fixed joint if the base should be ridigly attached.

rrbot.xacro

```
<inertial>
    <origin xyz="0 0 ${height1/2}" rpy="0 0 0"/>
    <mass value="${mass}"/>
    <inertia ixx="${mass / 12.0 * (width*width + height1*height1)}"
             ixy="0.0" ixz="0.0" iyy="${mass / 12.0 * (height1*height1 + width*width)}"
             iyz="0.0" izz="${mass / 12.0 * (width*width + width*width)}"/>
</inertial>
<xacro:include filename="$(find rrbot_description)/urdf/rrbot.gazebo"/>
<link name="world"/>

<joint name="fixed" type="fixed">
  <parent link="world"/>
  <child link="link1"/>
</joint>
```

rrbot.gazebo

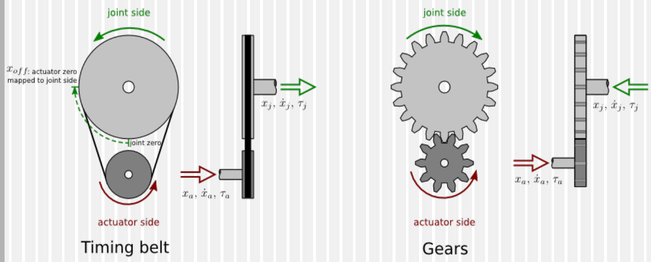
```
<gazebo reference="link2">
  <mu1>0.2</mu1>
  <mu2>0.2</mu2>
  <material>Gazebo/Black</material>
</gazebo>
```

URDF extension for ros_control (I): transmissions

- In order to use ros_control in a robot defined with URDF, we have to add **<transmission>** elements for linking actuators ⇔ joints that contain:
  - **<type>:** Type of transmission: Simple Reduction Transmission, Differential Transmission, Four Bar Linkage Transmission. In Gazebo, only "transmission_interface/SimpleTransmission".
  - **<joint>**: Name of the joint that the transmission is connected to.
    - **<hardwareInterface>:** Specifies joint-space hardware interface (EffortJointInterface in Gazebo)
  - **<actuator>:** Name of the actuator that the transmission is connected to.
    - **<mechanicalReduction>:** (Optional) Mechanical reduction at transmission.
    - **<hardwareInterface>:** Specifies joint-space hardware interface (not required after Gazebo-Indigo)

```xml
<transmission name="tran1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="joint1">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor1">
    <hardwareInterface>EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```

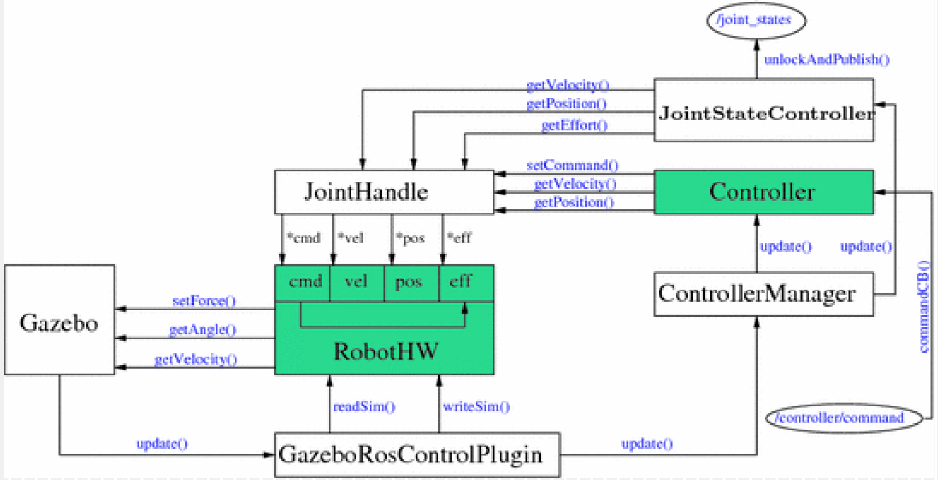
rrbot.xacro



	Effort	Velocity	Position
Actuator to joint	$\tau_j = n\tau_a$	$\dot{x}_j = \dot{x}_a/n$	$x_j = x_a/n + x_{off}$
Joint to actuator	$\tau_a = \tau_j/n$	$\dot{x}_a = n\dot{x}_j$	$x_a = n(x_j - x_{off})$

URDF extension for ros_control (II): Gazebo Plugin

- A Gazebo plugin needs to be added in the URDF for :
  - Parsing the transmission tags from the URDF
  - Loading the appropriate hardware interfaces in RobotHW (DefaultRobotHWSim)
  - Loading controller manager

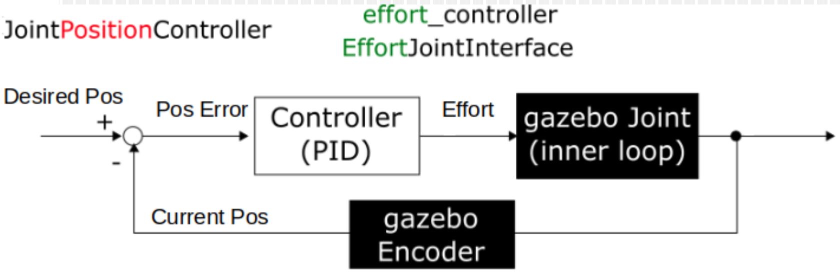



```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/rrbot</robotNamespace>
    <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
  </plugin>
</gazebo>
```

rrbot.gazebo

Complete ros_control-based package

- PID gains and controllers settings are saved in a **yaml file** (config subfolder):




```
rrbot:
  # Publish all joint states
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50
 # Position controllers
 joint1_position_controller:
   type: effort_controllers/JointPositionController
   joint: joint1
   pid: {p: 100.0, I: 0.01, d: 10.0}
...
```

rrbot_control.yaml

- Launch file:
  - 1. Load YAML
  - 2. Load controllers
  - 3. Load Robot State publisher (tf)

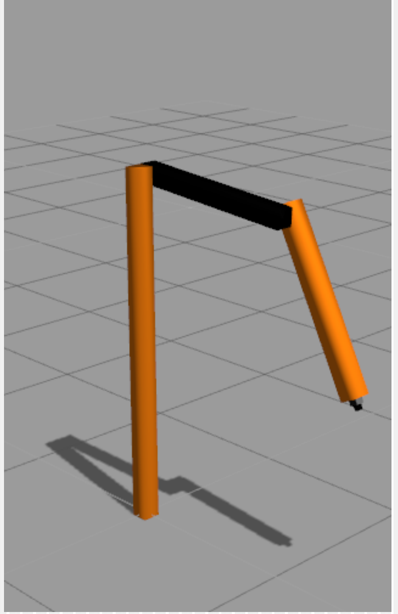
```
<launch>
  <rosparam file="$(find rrbot_control)/config/rrbot_control.yaml" command="load"/>

  <node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false" output="screen" ns="/rrbot" args="joint_state_controller
joint1_position_controller joint2_position_controller "/>

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher" respawn="false" output="screen"/>
   <remap from="/joint_states" to="/rrbot/joint_states" /></node>
</launch>
```

rrbot_control.launch

Example of RRbot

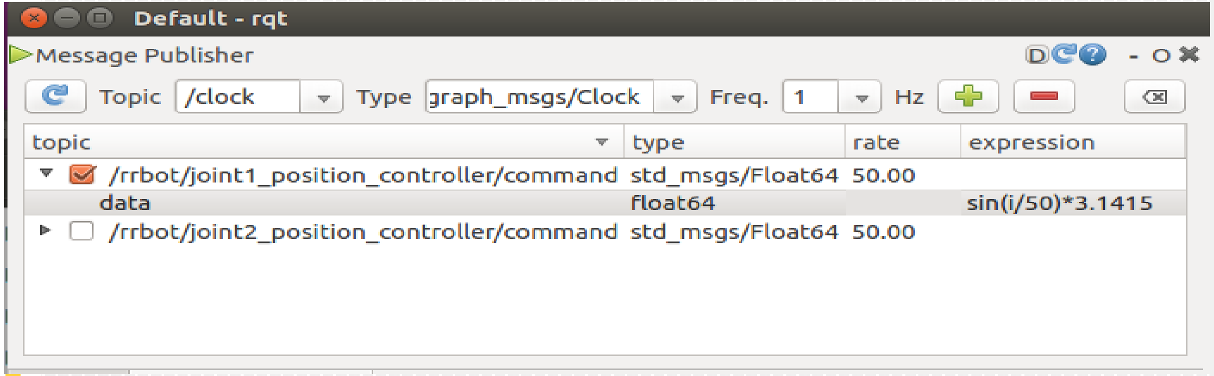


- 3 Packages:
  - **/rrbot_description:** URDF + xacro files.
  - **/rrbot_gazebo**: worlds + launch files for Gazebo.
  - **/rrbot_control:** YAML files + launch for controllers.
- Execute launch files to initialize system:
  - Initialize Gazebo (loads URDF in param/Gazebo) :
    roslaunch rrbot_gazebo rrbot_world.launch
  - Initialize controllers (loads YAML, controllers and State Publisher) :
    roslaunch rrbot_control rrbot_control.launch
    If controllers not found: sudo apt-get install ros-kinetic-ros-control ros-kinetic-ros-controllers ros-kinetic-gazebo-ros-control
- Send commands to controllers of joints:

rostopic pub -1 /rrbot/joint1_position_controller/command std_msgs/Float64 "data: 1.5"
rostopic pub -1 /rrbot/joint2_position_controller/command std_msgs/Float64 "data: 1.0"

Tuning PID control gains (I)

- Start rqt_gui : rosrun rqt_gui rqt_gui
- Add 2 message publishers (Plugins/Topics) for commands of joints 1 and 2:
  /rrbot/joint1_position_controller/command /rrbot/joint2_position_controller/command



`sin(i/rate*speed)*diff + offset`

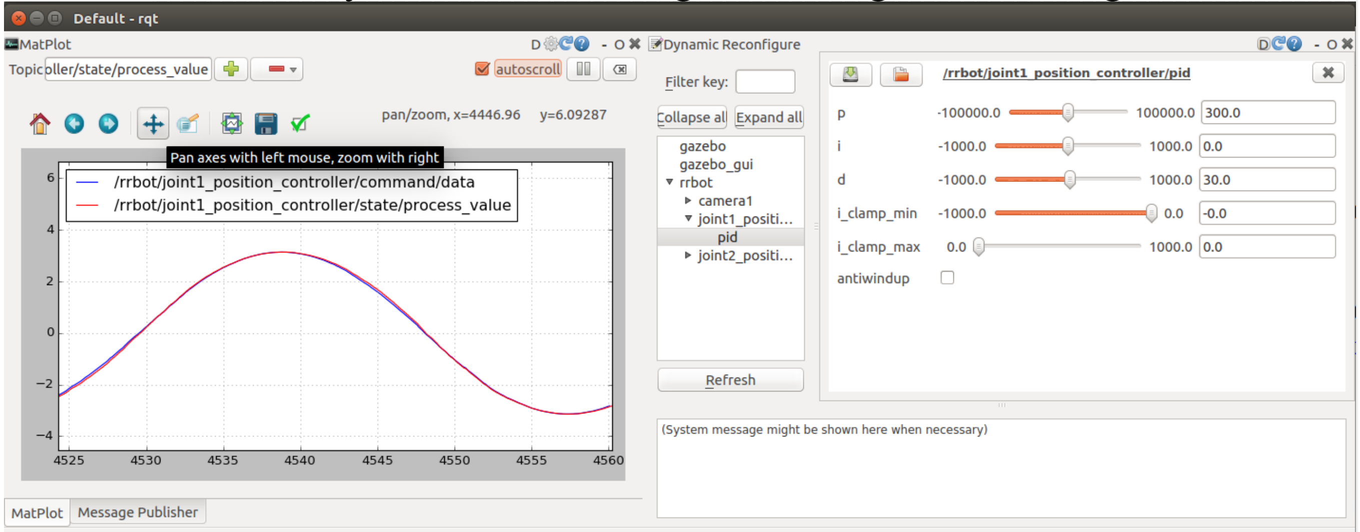
i - the RQT variable for time
rate - the frequency that this expression is evaluated (50 Hz).
speed - how quick you want the join to actuate. Start off with just 1 for a slow speed
upper_limit and lower_limits - the joint limits (-pi and +pi).
diff = (upper_limit - lower_limit)/2
offset = upper_limit-diff

- Change frequency to 50Hz and send 0 command to both joints
- Generate sinus command data for joint 1 → Expresssion: sin(i/50)*3.1415

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes

Tuning PID control gains (II)

- Add plot for comparing command and state (Plugins/Visualization)
  /rrbot/joint1_position_controller/command/data /rrbot/joint1_position_controller/state/process_value
- Add dynamic_reconfigure (Plugins/Configuration) for tuning pid gains :



PID TUNING PROCEDURE

0. Fix small value for Kp (10) and 0 for Kd/i

1. Increase Kp as high as you can for matching command/state without inducing wild oscillation

2. Increase Kd to remove overshoot

3. Adjust Ki to remove any residual offset

GOAL: Get the loop to settle as quickly as possible with as little overshoot as possible

- Use pan/zoom tool of plot (after disabling “Autoscroll”) for improving scale

INSTITUT PASCAL
sciences de l'ingénierie et des systèmes