LEONARDO

LEONARDO TECHNICAL TRAINING

# Robotic Software
# Lezione 4

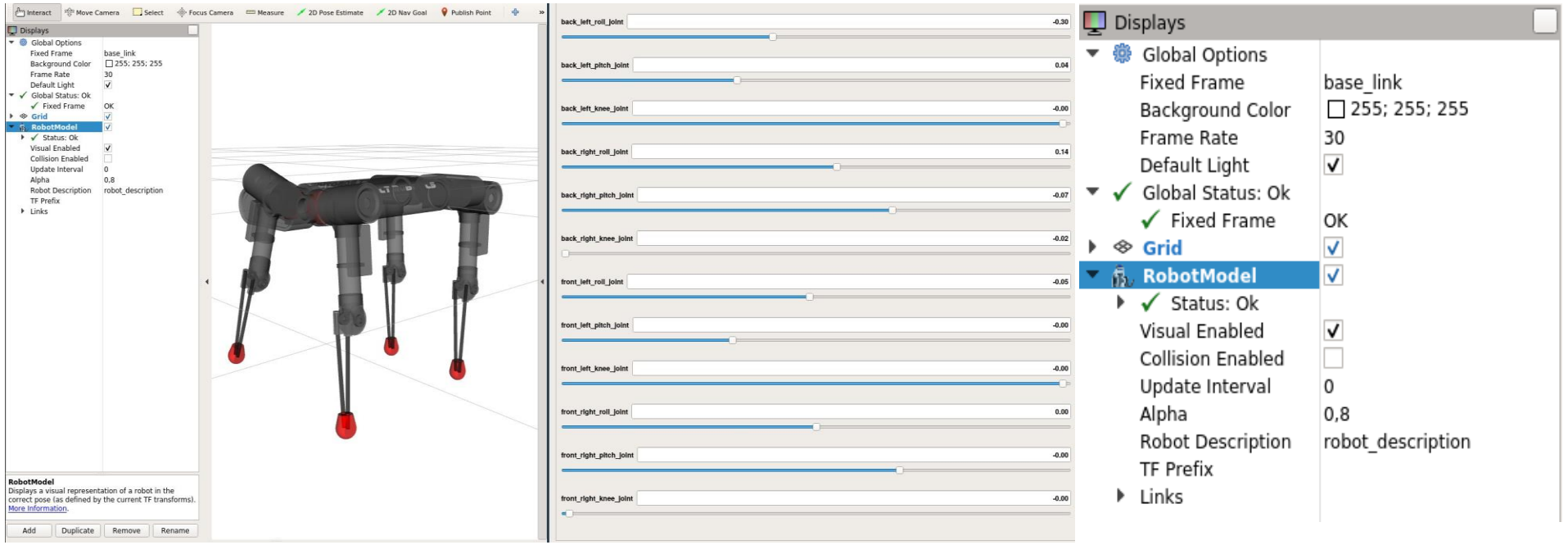
## ROS: Modeling, visualization and simulation

# Modeling and visualization

- Learn how to model a robotic structure from scratch:
  - Create a configuration file specifying the kinematics and the dynamics of your robot
  - Define and connect joints and links Define the shape of robotic links
  - Learn how to use a robot model for visualization and control
- ROS is so much useful in robot programming when you need functionalities already implemented in other ROS packages.
  - Several robotic software need for the knowledge of robotic structure of the robot to work:
  - forward / inverse kinematics motion and path planning obstacle avoidance navigation
- Model a robot from scratch is not easy
  - Robots can be composed by several joints and links
  - Links can be complex to design
  - How to characterize the dynamics?
  - Generate the robot model from the CAD
- **URDF** (Unified Robot Description Format) is the most popular file to model a robot

# Modeling and visualization

- RViz is used to visualize robot models. Start RViz node  using this command:
  - $ rosrun rviz rviz
- RViz has multiple configuration panels. The most useful  one is the Display panel.
- RViz has several plugins. Among them RobotModel is used  to visualize the robot in the visualization environment.



# Modeling and visualization

- XML documents must contain one root element that is the  parent of all other elements:
  - <root>
  - <child>
    - < subchild > . . . . . < / subchild >
  - </child>
  - </root >A tag could also contain attributes:
  - <child attr1="value"  attr2="value">  . . .  </child>

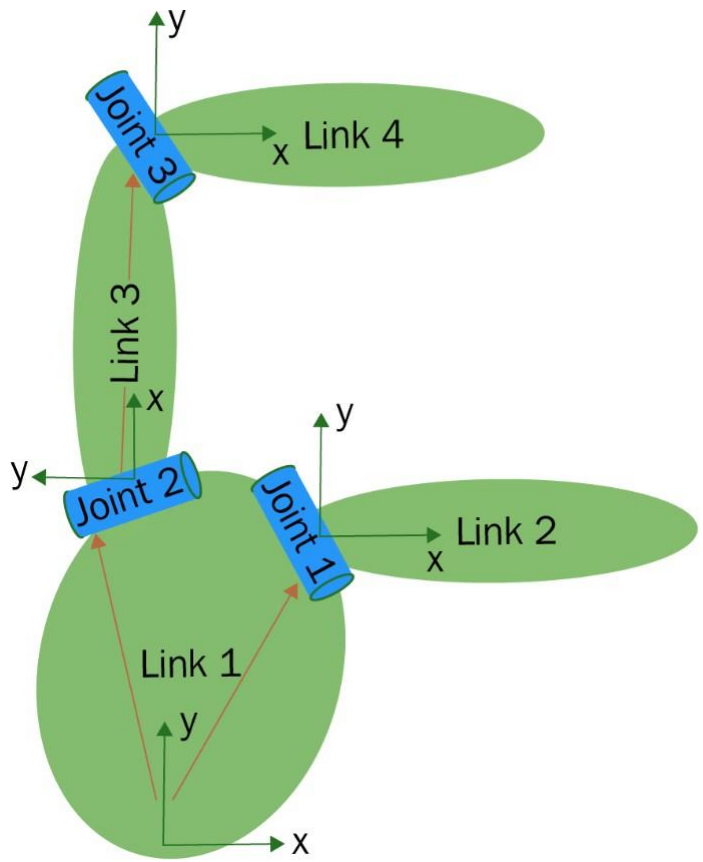
- A tag can also contain ONLY attributes
  - <child  attr1="value"  attr2="value">  . . .  </child>

# Modeling and visualization

```xml
<?xml version="1.0"?>
<robot name="pan_tilt">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.01" radius="0.2"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <material name="yellow">
        <color rgba="1 1 0 1"/>
      </material>
    </visual>

    <collision>
      <geometry>
        <cylinder length="0.03" radius="0.2"/>
      </geometry>
      <origin rpy="0 0 0" xyz="0 0 0"/>
    </collision>
    <inertial>
  <mass value="1"/>
  <inertia ixx="1.0" ixy="0.0" ixz="0.0" iyy="1.0" iyz="0.0" izz="1.0"/>
    </inertial>
  </link>

  <joint name="pan_joint" type="revolute">
    <parent link="base_link"/>
    <child link="pan_link"/>
    <origin xyz="0 0 0.1"/>
    <axis xyz="0 0 1" />
    <limit effort="300" velocity="0.1" lower="-3.14" upper="3.14"/>
    <dynamics damping="50" friction="1"/>
```



# Modeling and visualization

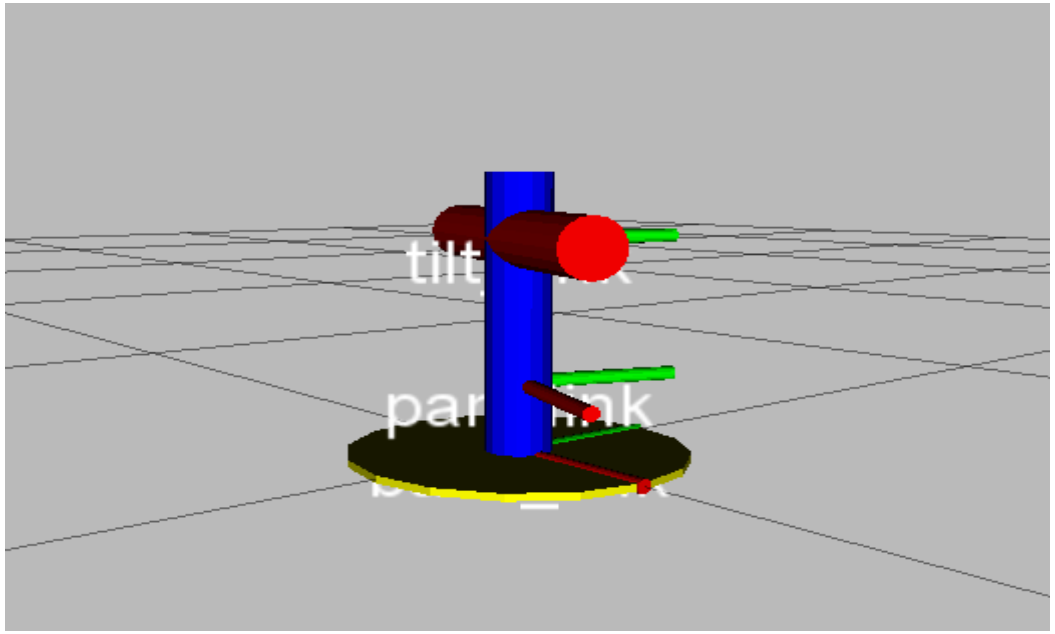
- robot : This tag encapsulates the entire robot model. Inside the robot tag, we can define the name of the robot, the links, and the joints of the robot
  - <robot name="<name of the robot>"
    - <link> . . . . . </link>
    - <joint> . . . . . </joint>
  - </robot>
- link : The link tag represents a single link of a robot defining its size, shape, and color. We can also specify the dynamic properties of the link
  - <link name="<name of the link>">
    - <inertial> . . . . . </inertial>
    - <visual> . . . . . </visual>
    - <collision> . . . . . </collision>
  - </link>
- The Visual section contains info about link shape
- The collision section describes the area surrounding the real link

# Modeling and visualization

- joint: here we can specify the kinematics and the dynamics  and the limits of the joint.
- Different type of joints are supported: revolute, continuous,  prismatic, fixed, floating, and planar.
- <joint name="name of the joint" />
  - <parent link="name of the parent link" />
  - <child link="name of the child link" />
  - <limit position, velocity, effort />
- </joint>
- A joint is formed between two links; the first is called the  Parent link, and the second is called the Child link.

# Example 1.4

- Robot models are included in dedicated packages in order to invoke the configuration and modelling files when needed
- Develop a ROS package containing the model of a 2 DOF robot
  - Goal: model a robot with two degree of freedom
  - A degree of freedom is a controllable joint of our robot
    - Pan: rotation left − > right
    - Tilt: rotation up − > down

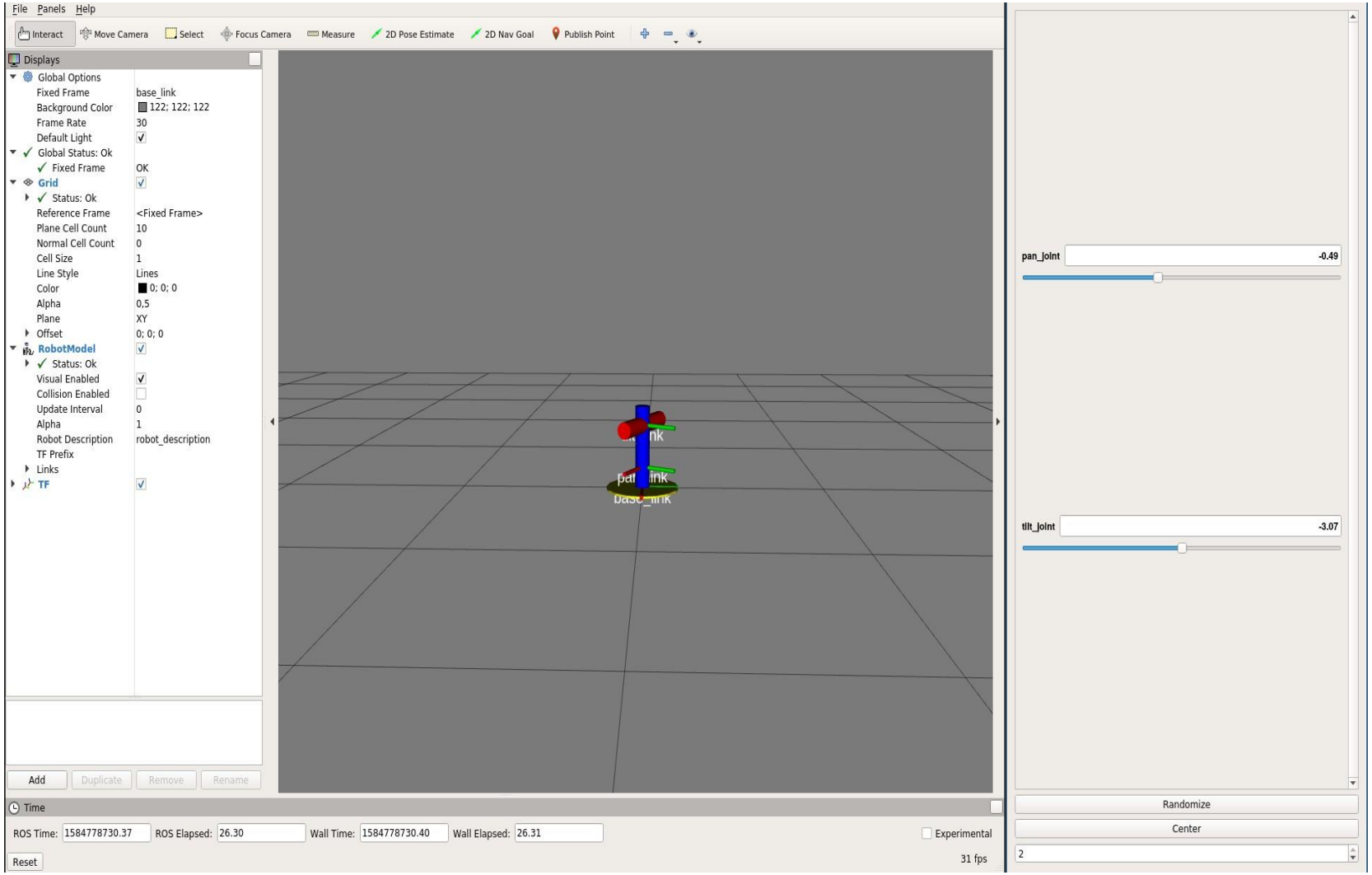


# Example 1.4

- Create a ROS package to include all the necessary files used to setup the simulated model
  - $ roscd
  - $ cd ../src
  - $ catkin_create_pkg robot_description_pkg urdf rviz
- Typically, description files are included in the urdf subfolder of the ROS package. This is optional (not like messages, services or actions)
- To visualize the robot in RVIZ, launch files can be used
- A simple gui can be used also to move the joints of our robot

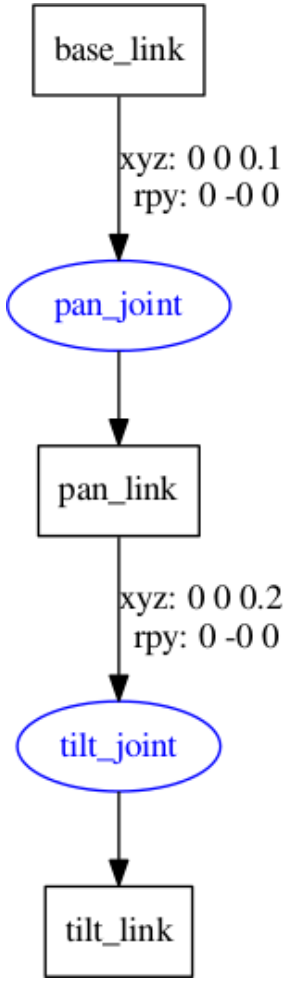
# Example 1.4

- Visualize the URDF model of the robot in RVIz



# check_urdf

- check_urdf accepts in input the file to check
  - $  roscd robot_description_pkg/urdf/
  - $ check_urdf pan_tilt.urdf
- We can visualize the structure of the robot graphically:
  - $ urdf_to_graphiz  pan_tilt.urdf
- The robot model can be visualized using RViz  RViz shows the shape of the robot
- Using RViz we can test the connections between the links  RViz uses RobotModel plugin to display the robot
- The robot model is taken from ROS parameter server using  the robot description parameter



# Tf system

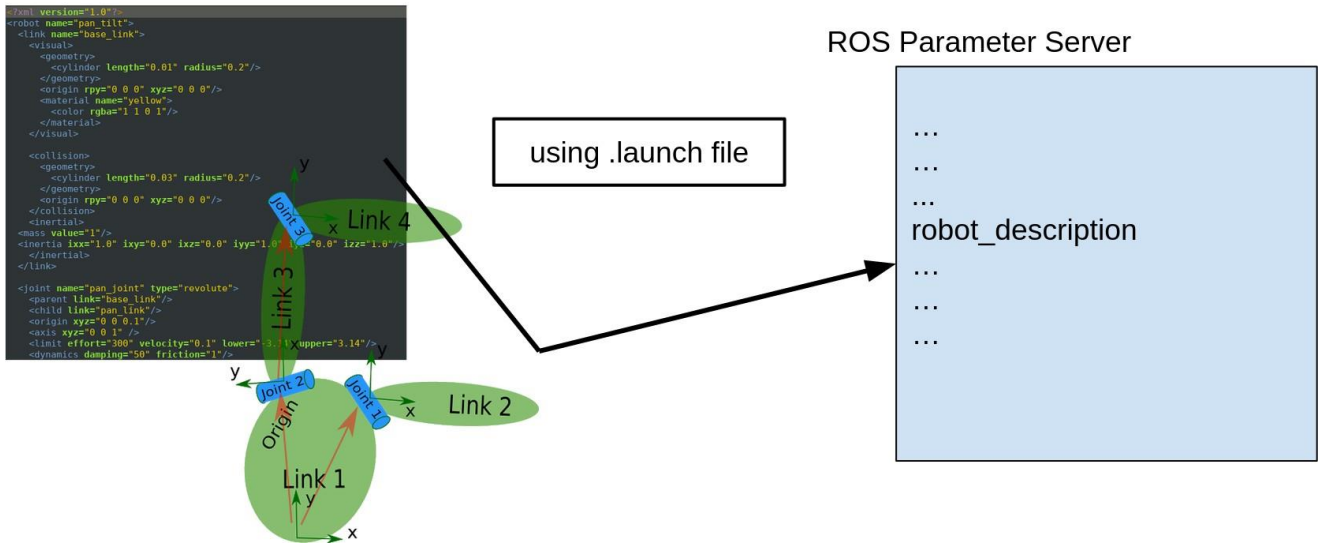
- Visualize the URDF model of the robot in RVIz
- What is behind the visualization process of RViz?
  - Same information needed by RViz visualization are used by all the other external packages!!!
- The configuration of a manipulator is needed to allow grasping
- The full pose of a mobile robot inside its environment is needed for smart navigation
- Robotic systems typically have many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc…
- tf package keeps track of all these frames over time, and allows developers to develop queries to retrieve the relative pose of a frame with respect to the other frames
- How is oriented the head of the head of the robot with respect to its base frame?
- What is the pose of the object in my gripper relative to my base?
- What is the current pose of the base frame in the map frame?

# Tf system

- The chain to retrieve information from our robot model:
  - joint_state_publisher uses URDF to get information about the name and the type of each joint, then it publish a joint_states message containing such values.
  - robot_state_publisher uses the URDF model to get information about the kinematic structure of the robot and the joint_states message to fill the tf tree.
  - RViz uses the URDF to display the robot model, and the tf tree to display the relative position of each joint of the robot.

# URDF to Parameter server

Unified Robot Description Format (URDF)



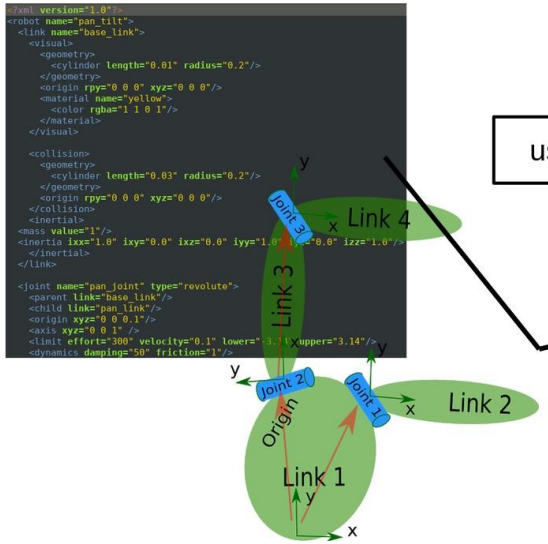

using .launch file

ROS Parameter Server

…
…
…
robot_description
…
…
…

# URDF to Parameter server

Unified Robot Description Format (URDF)



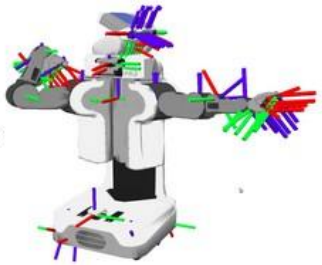

using .launch file

ROS Parameter Server

…
…
…
robot_description
…
…
…

ROS Parameter Server

…
…
…
robot_description
…
…
…



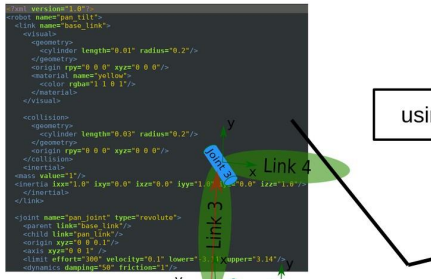
**robot_state_publisher:** calculate the forward kinematics of the robot and publish the results via tf.

launch file: we can configure the name of the parameter to use into the robot_state_publisher node

```
<node …>
      robot_description
</node>
```

# URDF to Parameter server

Unified Robot Description Format (URDF)

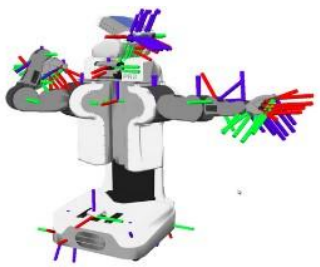


using .launch file

ROS Parameter Server

…
…
…
robot_description
…
…
…



Topics:

…
…
…
/joint_states
…
…
…

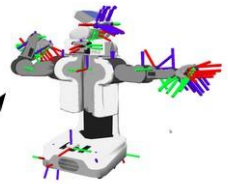
**robot_state_publisher:** calculate the forward kinematics of the robot and publish the results via tf.

Joint position???

**joint_state_publisher**: The package reads the robot_description parameter from the parameter server, finds all of the non-fixed joints and publishes a JointState message with all those joints defined.

ROS Parameter Server

…
…
…
robot_description
…
…
…



**robot_state_publisher:** calculate the forward kinematics of the robot and publish the results via tf.

launch file: we can configure the name of the parameter to use into the robot_state_publisher node

```
<node …>
        robot_description
</node>
```

# XACRO format

- URDF structure is not flexible when you need to create a complex robot model.
- missing of simplicity, reusability, modularity and programmability
- If someone wants to reuse a URDF block 10 times in his robot description, he must copy and paste it 10 times.
- If there is an option to use this code block and make multiple copies with different settings
  - xacro is a powerful file description format that aims at: Simplify URDF : The xacro is the cleaned-up version of URDF. It creates macros inside the robot description and reuses the macros.
- Increase robot model programmability: the xacro language supports a simple programming statement in its description.
  - xacro is a powerful file description format that aims at: xacro file format represents an updated version of URDF. URDF and xacro have the same power of expression
- You can always convert a xacro file in an URDF one and vice-versa.

# XACRO format

- Property: declare variables to change the constants of your robot model.
  - <xacro:property name="base_link_length" value="0.1" />
    - You can also fit properties into blocks:
  - <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>

- Using xacro file you can also use math expressions:
  - <xacro:property name="radius" value="4.3" />
    - <circle diameter="${2 * radius}" />
  - and conditional blocks:
    - <xacro:if value="<expression>">
      - <... some xml code here ...>
    - </xacro:if>
    - <xacro:unless value="<expression>">
      - <... some xml code here ...>
    - </xacro:unless>

# XACRO format

- The most important element of xacro format is the  possibility to define macro blocks.
  - <xacro:macro name="inertial_matrix" params="mass">
    - <inertial>
      - <mass value="${mass}" />
      - <inertia ixx="0.5" ixy="0.0" ixz="0.0"  iyy="0.5" iyz="0.0" izz="0.5" />
    - </inertial>
  - </xacro:macro>

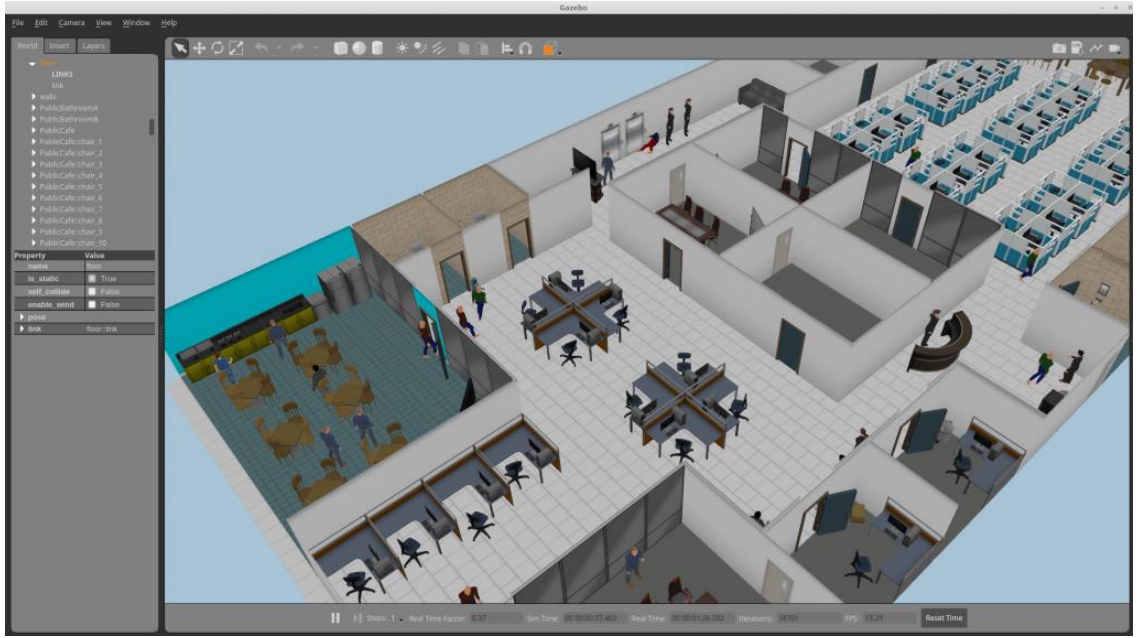
- We can replace each inertial code with a single line, as given  here:
  - <xacro:inertial_matrix mass="1"/>

# XACRO format

- xacro definition improves the code readability and reduces the number of lines
- All ROS software require an URDF file.
- So?
  - After designing the xacro file, we can use the following command to convert it to a URDF file:
    - $ rosrun xacro xacro pan_tilt.xacro > pan_tilt_generated.urdf
  - Be careful to refer to your main xacro file, and not to that one containing the definition of the macro blocks.
  - Is possible to directly refer to the xacro file?
    - In display_xacro.launch file we set the robot description parameters with the content of a textfile.
  - We can do the same filling such variable with the content of a command
    - <param name="robot_description" command="$(find xacro)/xacro $(find robot_description_dir)/urdf/pan_tilt.xacro" />

# Gazebo simulator

- Multi-robot simulator for complex indoor and outdoor robotic simulations
- Gazebo has simulation models of popular robots, sensors, and a variety of 3D objects in their repository (https://bitbucket.org/osrf/gazebo_models/)
- Gazebo has a good interface in ROS and ROS2, which exposes the whole control of Gazebo in ROS
- Gazebo is a standalone software
  - can be installed also without ROS



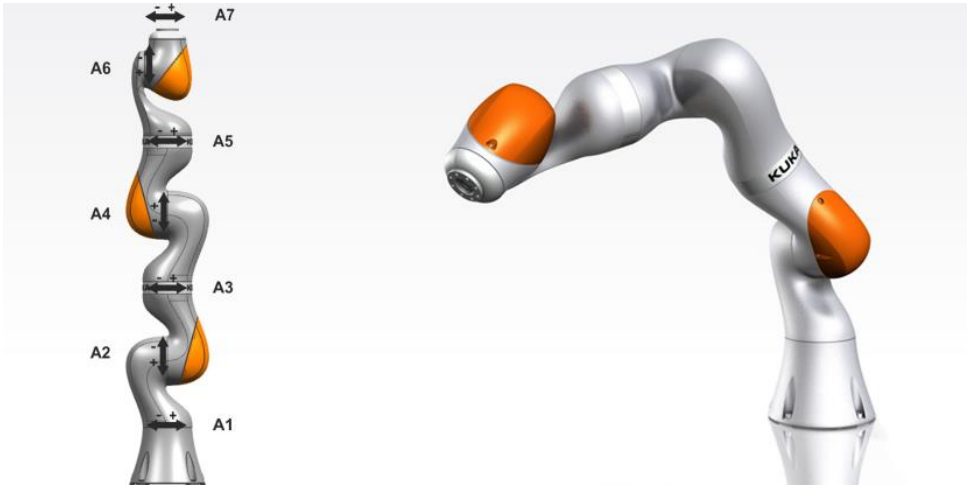

GAZEBO

# Gazebo simulator

- The default version installed from Noetic ROS is Gazebo 11.0 and it is interfaced with ROS thanks to the following packages:
  - gazebo_ros_pkgs: This contains wrappers and tools for interfacing ROS with Gazebo
  - gazebo-msgs: This contains messages and service data structures for interfacing with Gazebo from ROS
  - gazebo-plugins: This contains Gazebo plugins for sensors, actuators, and so on
  - gazebo-ros-control: This contains standard controllers to communicate between ROS and Gazebo
- To check if Gazebo is properly installed use the following commands:
  - $ roscore
  - $ rosrun gazebo_ros gazebo
- These commands will open the Gazebo GUI
- $ rostopic list
  - /gazebo/model_states contains geometrical information about the model spawned into the scene
  - This message is part of gazebo_ros messages
  - /gazebo/model_states topic contains the exact pose of your models like a sort of oracle.

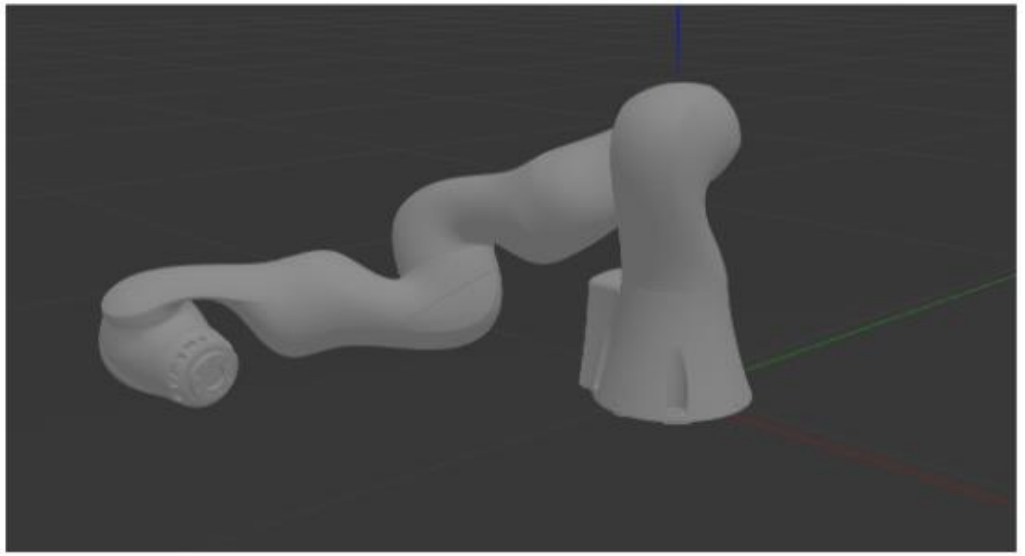
# Example 2.4

- Configure your robot to work with the simulation environment
- Goal: Import a kuka iiwa industrial robot in the Gazebo world
    - $ git clone https://github.com/robotic-software/kuka_iiwa_support



# Example 2.4

- The robot is not able to contrast the gravity, since it has no motor controller defined in its model.
- The robot can not be actuated



- Add controllers to the robot

# Example 2.4

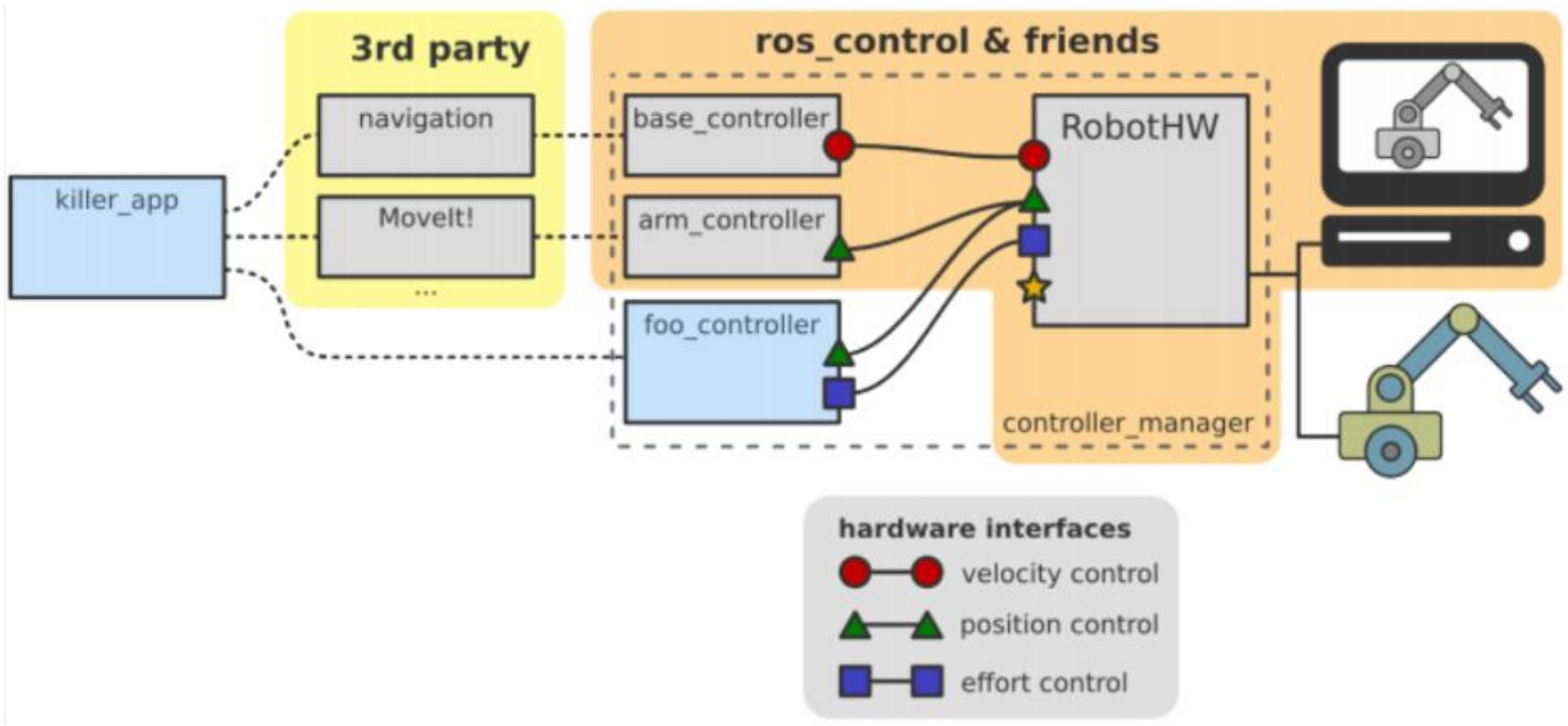
- The <plugin> tag needs the name of the plugin to load libgazebo_ros_control.so
- The <robotNamespace> element can be given as the name of the robot
- If we are not specifying the name, it will automatically load the name of the robot from the URDF.
- The considered hardware interfaces in ROS are
  - JointStateInterface
  - PositionJointInterface
  - EffortJointInterface
  - VelocityJointInterface
- The <legacyModeNS> enables the compatibility with old version of gazebo_ros_control interface

# Example 2.4

- At this point, the robot is able to react to the world gravity
- It will remain stable on the initial position (the candle position)
- We can not control it
- Let's see how to enable the possibility to control the joints of the robot using the ros_control package
- To move each joint of the kuka iiwa, we need to assign a ROS controller
- For each joint we need to attach a controller that is compatible with the hardware interface mentioned inside the transmission tags of the xacro file
- A ROS controller consists of a feedback mechanism that can receive a set point and control the output using the feedback from the actuators.
- ROS controller interacts with the hardware using the hardware interface
- The hardware interface acts as a mediator between ROS controller output and the real or simulated hardware
  - It allocates the resources to control it considering the data generated by the ROS controller
  - We can deploy the same software structure without carrying out any modification to our source code previously tested in the simulation scene

# ROS Controllers

- We also can used different type of ROS controllers:
  - joint_position_controller: This is a simple implementation of the joint position controller.
  - joint_state_controller: This is a controller to publish joint states.
  - joint_effort_controller: This is an implementation of the joint effort (force) controller



# ROS Controllers

- The hardware interface is decoupled from actual hardware and simulation
- The values from the hardware interface can be fed to Gazebo for simulation or to the actual hardware itself
- The hardware interface is a software representation of the robot and its abstract hardware
- The resource of the hardware interfaces are actuators, joints, and sensors
- Some resources are read-only: joint states, IMU, force-torque sensors, and so on
- Some resources are read and write compatible: position, velocity, and effort joints.

# ROS Controllers

- Interfacing robot controllers to each joint is a simple task, only two steps are needed:
  - Configure the ROS controllers using a proper configuration file (yaml format)
  - Load the configuration file and launch the ROS controller node using a proper launch file.
- The first task is to write a configuration file for desired controllers.
- We want to start two controllers: the joint state controller/JointStateController controller that provides information about the joint state
- A set of position controllers/JointPositionController to control the position of each robot joint

# ROS Controllers

- The first task is to write a configuration file for desired controllers.
- We want to start two controllers:
  - the joint state controller/JointStateController controller that provides information about the joint state
  - A set of position controllers/JointPositionController to control the position of each robot joint1

# KDL

- KDL stands for Kinematics and Dynamics Library. This library is born with the OROCOS (Open Robots Control Software) project and can be used to solve different kinematic and dynamic problems.
- The Kinematics and Dynamics Library (KDL) develops an application independent framework for modeling and computation of kinematic chains, such as robots, biomechanical human models, computer-animated figures, machine tools, etc.
- It provides class libraries for geometrical objects (point, frame, line,... ), kinematic chains of various families (serial, humanoid, parallel, mobile,... ), and their motion specification and interpolation.

# KDL

- Kinematics and Dynamics of kinematic chains: You can represent a kinematic chain by a KDL Chain object and use KDL solvers to compute anything from forward position kinematics, to inverse dynamics. The kdl parser includes support to construct a KDL chain from a XML Robot Description Format (URDF) file.

- Kinematics of kinematic trees: You can represent a kinematic chain by a KDL Chain object, and use KDL solvers to compute forward position kinematics.

- Examples:
  - Kinematic: in the first example, we will see how KDL can be used to solve forward and inverse kinematics
  - Dynamic: in the second example, we use KDL to calculate dynamic parameters of the robotic manipulator and
    generate the force to control it.
- However, both examples need to know the model of the robot and it's passed to KDL framework initializing a KDL Tree. We will see in the next section how to create it using the URDF file.

# KDL

- kdl parser is a ROS package providing an easy way to construct a full KDL Tree object. In particular, this Tree could be build manually specifying Joints and Links or a using the URDF xml description file
- The Tree is initialized directly from the URDF file that is loaded into the ROS parameter server. As usual we can use the launch file to fill the robot_description parameter

# Example 3.4

- Forward and inverse kinematics of the robot are calculate considering the position of its joints
- Using the forward kinematic we get the position of the robotic end effector Inverse kinematic solver is used to calculate the desired joint values to apply to bring the manipulator in a given location
- For this example, we will use a position-controlled robot. You can start the robot with the following command:
  - $ roscd
  - $ cd ../src
  - $ git clone https://github.com/robotic-software/lbr_iiwa_description.git
  - $ roslaunch lbr_iiwa_description gazebo_ctrl.launch

# Example 4.4

- Inverse Dynamics:
  - In some cases, we need to exploit the dynamic of the system to improve the performance of our controllers.
  - Inverse dynamics is a method for computing forces and/or torques based on the kinematics (motion) of a body and the body's inertial properties (mass and moment of inertia).
  - In robotics, inverse dynamics algorithms are used to calculate the torques that a robot's motors must deliver to make the robot's end-point move in the way prescribed by its current task.
  - We need a robot that can be controlled using effort: the hardware interface EffortJointInterface must be used for the iiwa simulation.
  - We need a robot that can be controlled using effort: the hardware interface EffortJointInterface must be used for the iiwa simulation. To start this new robot model, use the following command:
    - $ roslaunch lbr_iiwa_description gazebo_effort_controller.launch
    - This package needs for effort controllers
      - $ sudo apt-get install ros-melodic-effort-controllers
  - If we don't apply a force command to motor joints, the robot falls down to the floor.
  - In this case, we will not rely on a solver to implement the inverse dynamics controller, but directly calculate the body's inertial properties using KDL template and than implement a PD controller to generate the torque command

# Example 4.4

- Position error:

```
1       Eigen::VectorXd e = _initial_q->data - _q_in->data; //Keep
            initial position
```

- Velocity error:

```
1       Eigen::VectorXd de = -_dq_in->data; //Desired velocity: 0
```

- intertia matrix, the coriolis terms and the gravity temrs:

```
1       _dyn_param->JntToMass( *_q_in, jsim_);
2       _dyn_param->JntToCoriolis( *_q_in, *_dq_in, coriol_);
3       _dyn_param->JntToGravity( *_q_in, grav_);
```

# Differential drive robot

- Model and control a differential drive robot
  - https://github.com/robotic-software/robot_description_pkg.git
- Control the robot using ROS topics
  - https://github.com/robotic-software/key_teleop

- $ roslaunch robot_description_pkg spawn_diff_robot.launch
- $ rosrun key_teleop key_teleop

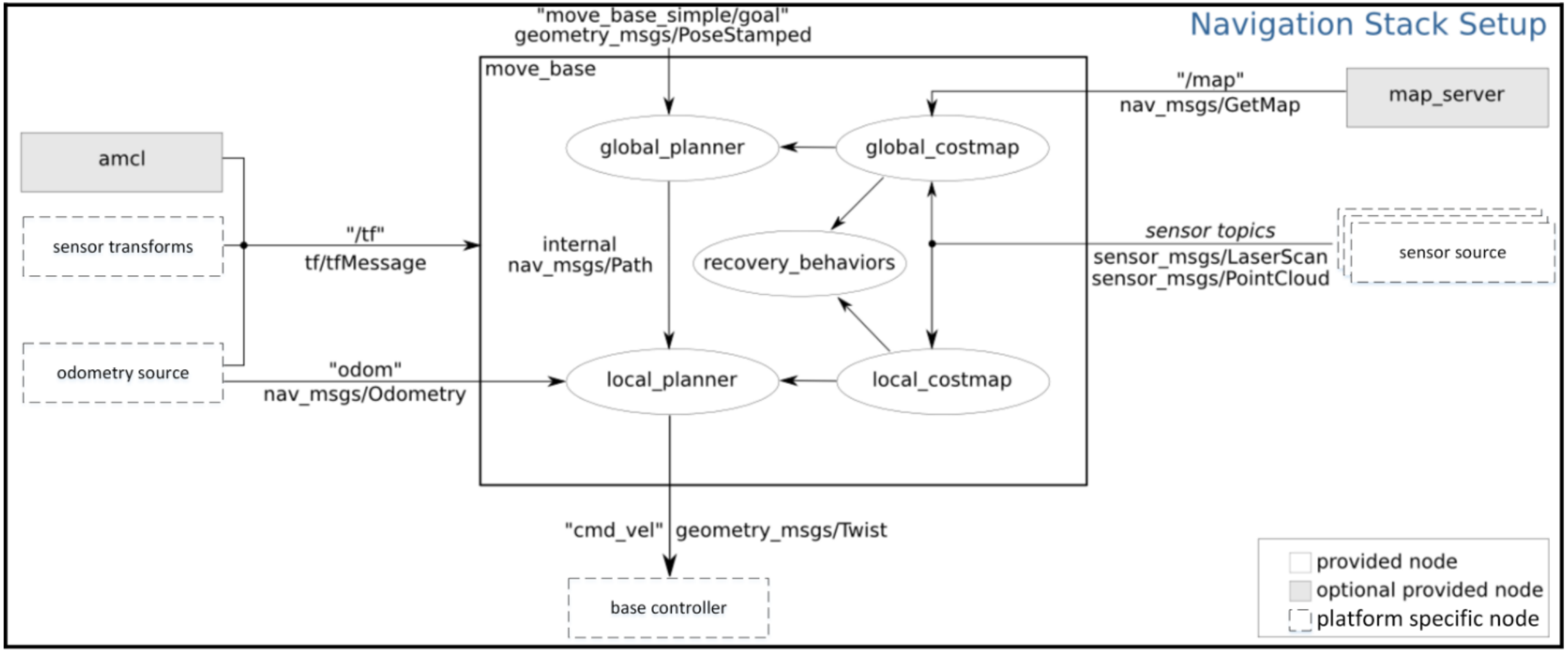
# Navigation stack

- Navigation stack contains a set of powerful tools and libraries to work mainly for mobile robot navigation.
- The Navigation stack contains ready-to-use navigation algorithms which can be used in mobile robots
  - especially for differential wheeled robots.
- Using these stacks, we can make the robot autonomous, and that is the final concept that we are going to see in the Navigation stack.
- The main aim of the ROS Navigation package is to move a robot from the start position to the goal position, without colliding with the environment.
- The ROS Navigation package comes with an implementation of several navigation-related algorithms which can easily help implement autonomous navigation in the mobile robots.
- The user only needs to feed the goal position of the robot and the robot odometry data from sensors such as wheel encoders, IMU, and GPS, along with other sensor data streams, such as laser scanner data or 3D point cloud from
  sensors such as depth sensor.
- The output of the Navigation package will be the velocity commands that will drive the robot to the given goal position.

# Navigation stack

- The ROS Navigation stack is generic.
- There are some hardware requirements that should be satisfied by the robot.
- The following are the requirements:
  - The Navigation package will work better in differential drive and holonomic (total DOF of robot equals to controllable DOF of robots).
  - The mobile robot should be controlled by sending velocity commands in the form of linear and angular velocity.
  - The robot should be equipped with a vision (rgb-d) or laser sensor to build the map of the environment.
  - The Navigation stack will perform better for square and circular shaped mobile bases. It will work on an arbitrary shape, but performance is not guaranteed.

# Navigation stack components



# Move base

- The move base node is from a package called move base.
- The main function of this package is to move a robot from its current position to a goal position with the help of other navigation nodes
- The move base node inside this package links
  - the global-planner and the local-planner for the path planning
  - Connecting to the rotate-recovery package if the robot is stuck in some obstacle
  - Connecting global costmap and local costmap for getting the map.

# Navigation stack

- Packages which are linked by the move base node:
  - global-planner: libraries and nodes for planning the optimum path from the current position of the robot to the goal position, with respect to the robot map
  - local-planner: navigate the robot in a section of the global path planned using the global planner. The local planner will take the odometry and sensor reading, and send an appropriate velocity command to the robot controller for completing a segment of the global path plan
  - costmap-2D: represent a map the robot environment
- The following are the other packages which are interfaced to the move base node:
  - map-server: the map-server package allows us to save and load the map generated by the costmap-2D package.
  - AMCL: AMCL (Adaptive Monte Carlo Localization) is a method to localize the robot in a map. This approach uses a particle filter to track the pose of the robot with respect to the map, with the help of probability theory.
    - In the ROS system, AMCL accepts a *sensor_msgs/LaserScan* to create the map.
  - gmapping: the gmapping package is an implementation of an algorithm called Fast SLAM, which takes the laser scan data and odometry to build a 2D occupancy grid map.

# Navigation stack

- To navigate unknown environments a robot must be able to build a map.
  - This process is called mapping.
- During the navigation the robot should also be able to localize during the map it is creating:
  - This process is called localization.
- These two steps could be made at the same time:
  - SLAM: simultaneous localization and mapping
- One of the most famous tools to do 2D SLAM is called gmapping.
- The ROS Gmapping package is a wrapper of the open-source implementation of the SLAM algorithm called OpenSLAM
- The package contains a node called slam gmapping, which has the aim to create a 2D occupancy grid map from the laser scan data and the pose of the mobile robot.
- Use gmapping on the differential drive robot:
  - it is already endowed with a laser scanner and the odometry data are provided by the differential drive robot plugin.

# Example 5.4

- Configure a differential drive robot to use the navigation stack
  - Localization & Mapping
  - Navigation
    - $ sudo apt-get install ros-melodic-gmapping
    - $ sudo apt-get install ros-melodic-move-base
    - $ sudo apt-get install ros-melodic-dwa-local-planner

- Create and save a map of the environment
  - Map server
    - $ sudo apt-get install ros-melodic-map-server

# Example 5.4

- Generate the pose of the robot
  - $ roslaunch robot_description_pkg gmapping.launch
- After getting the current position of the robot, we can send a goal position to the move base node.
  - $ roslaunch robot_description_pkg move_base.launch
- The move base node will send this goal position to a global planner, which will plan a path from the current robot position to the goal position.
- This plan is with respect to the global costmap, which is feeding from the map server.
- The global planner will send this path to the local planner, which executes each segment of the global plan.
- The local planner gets the odometry and the sensor value from the move base node and finds a collision-free local plan for the robot.
- The local planner is associated with the local costmap, which can monitor the obstacle(s) around the robot.
- Use RVIZ to visualize the state of the system and require a new command

# move_base

- Since move base implements some action server to allow the navigation of the robot, we can write an action client to perform the generation of the path.
- However, in our first example we can use RViz user interface to perform the robot navigation.
- Start writing the launch file for the move base package. Configuration files involved into the planning process.
  - costmap_common_params.yaml
  - global_costmap_params.yaml
  - local_costmap_params.yaml
  - dwa_local_planner_params.yaml
  - move_base_params.yaml

# Exercise 1.4

- Write a ROS node called ros_navigation consisting of one node. This node requests to the navigation stack the possibility to reach a waypoint. A total number of 4 waypoints should be considered

```
1  #include "ros/ros.h"
2  #include <move_base_msgs/MoveBaseAction.h>
3  #include <actionlib/client/simple_action_client.h>
4
5  int main( int argc, char** argv) {
6
7    ros::init(argc, argv, "move_base_client");
8    ros::NodeHandle nh;
9    actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction>
         ac("move_base", true);
10   ac.waitForServer(); //will wait for infinite time
11
12   move_base_msgs::MoveBaseGoal goal;
13   goal.target_pose.header.frame_id = "map";
14   goal.target_pose.pose.position.x = 5.0;
15   goal.target_pose.pose.orientation.w = 1.0;
16
17   ac.sendGoal(goal);
18   bool done = false;
19   ...
```