IDRA

Interdipertimental Robotics Institute
University of Trento

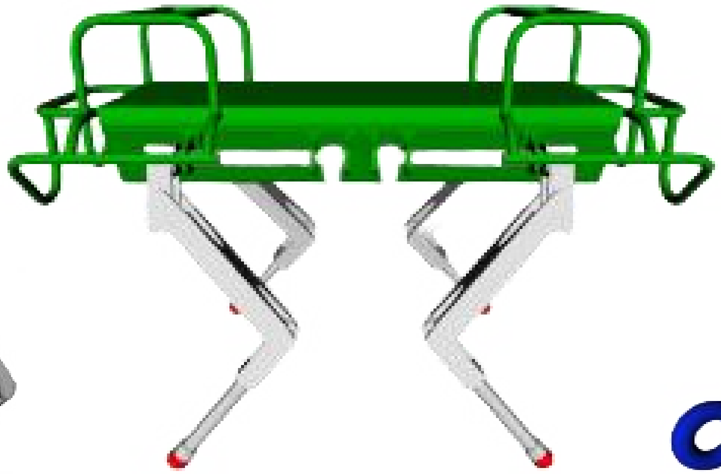
GAZEBO/ROS programming Lab

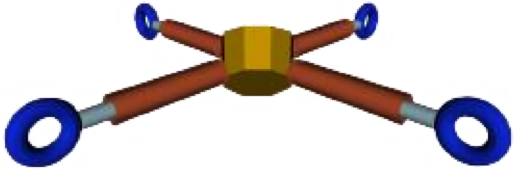
An Open-Source Cross-Platform Robotics Framework

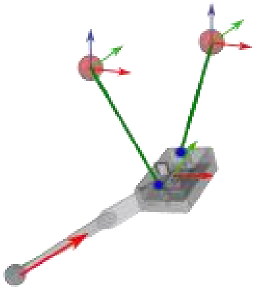
What is locosim?

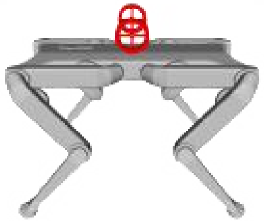
- Is a didactic framework environment to simulate both **fixed-base** and **floating-base** robots
- Is written in Python 3 and C++
- Already supported robots: HyQ, Ur5, Solo, Aliengo, Go1 (Quadrupeds), Jumpleg (fixed)
- Base class sckeleton to ease the simulation of new robots
- it has a few dependencies :
  - Robot Operating System (ROS),
  - Pinocchio Library
- Recently Locosim has been successfully tested in controlling the quadruped robot Aliengo at 500 Hz with real-time performances

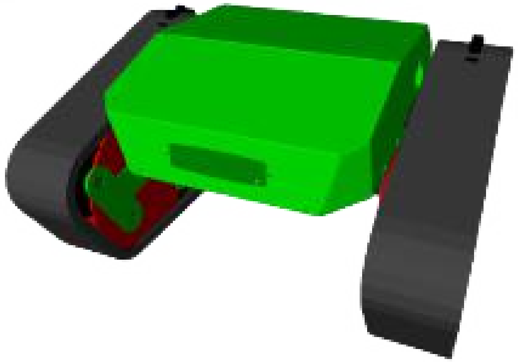

Unitree Robotics

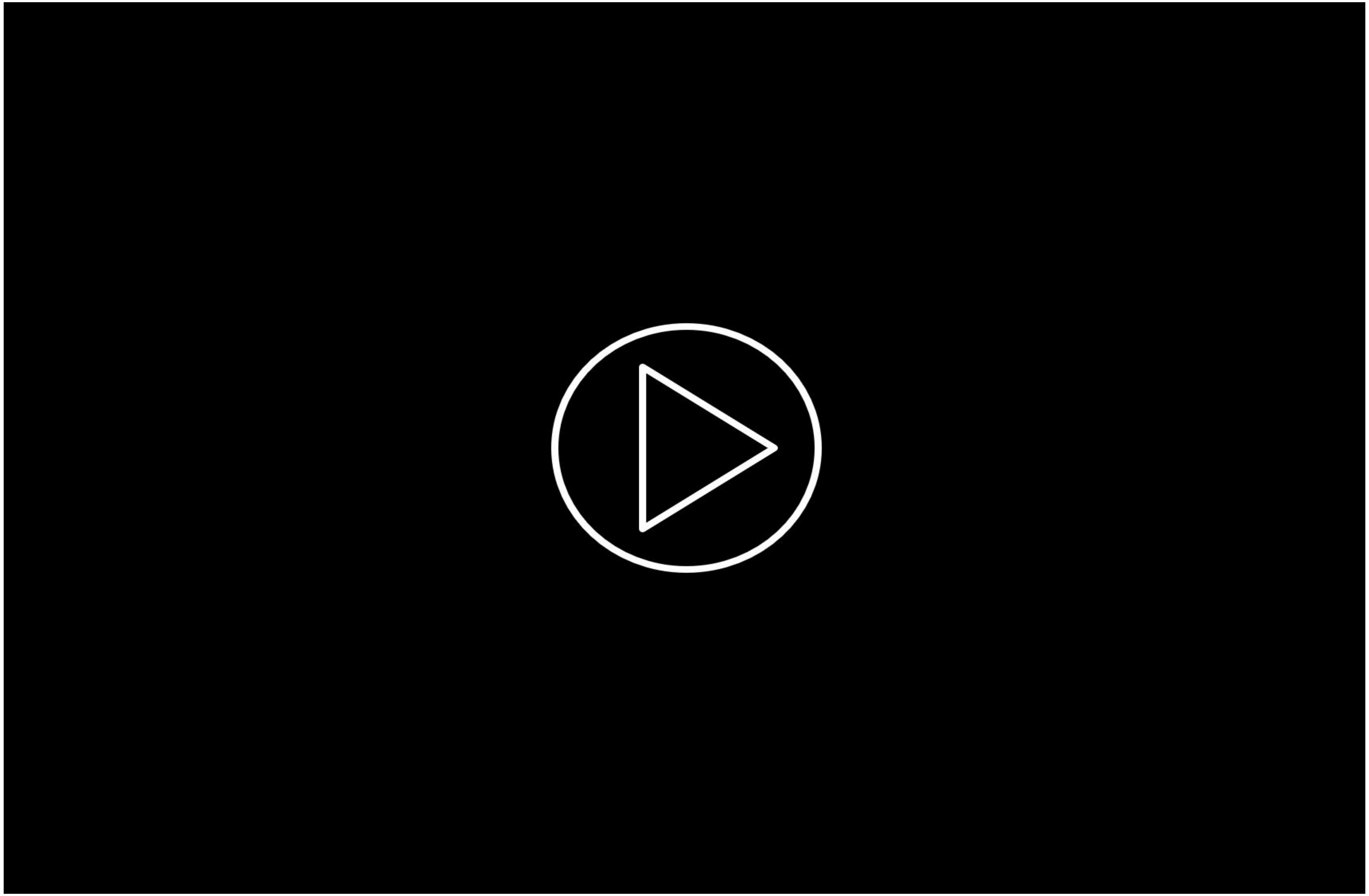




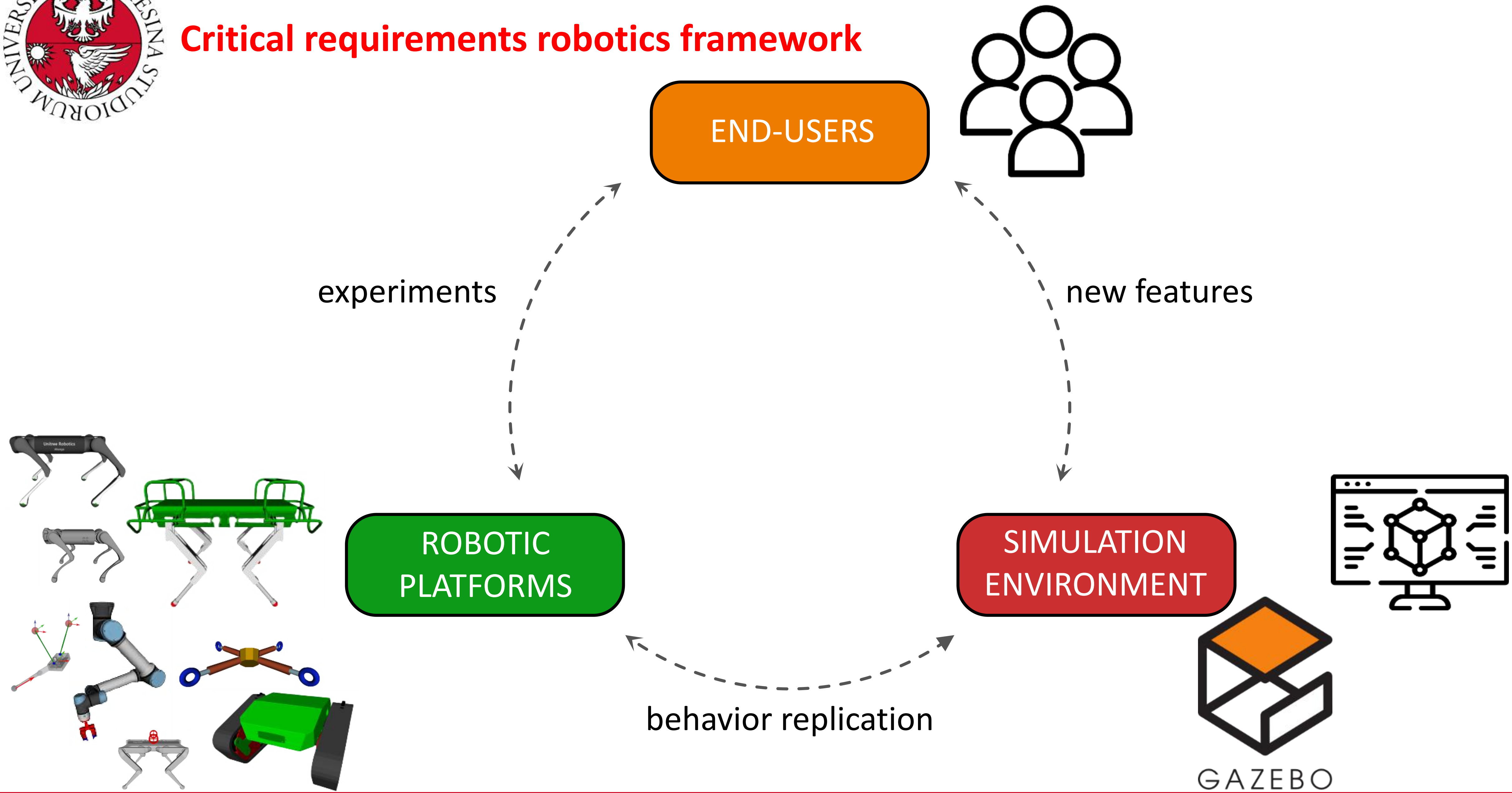


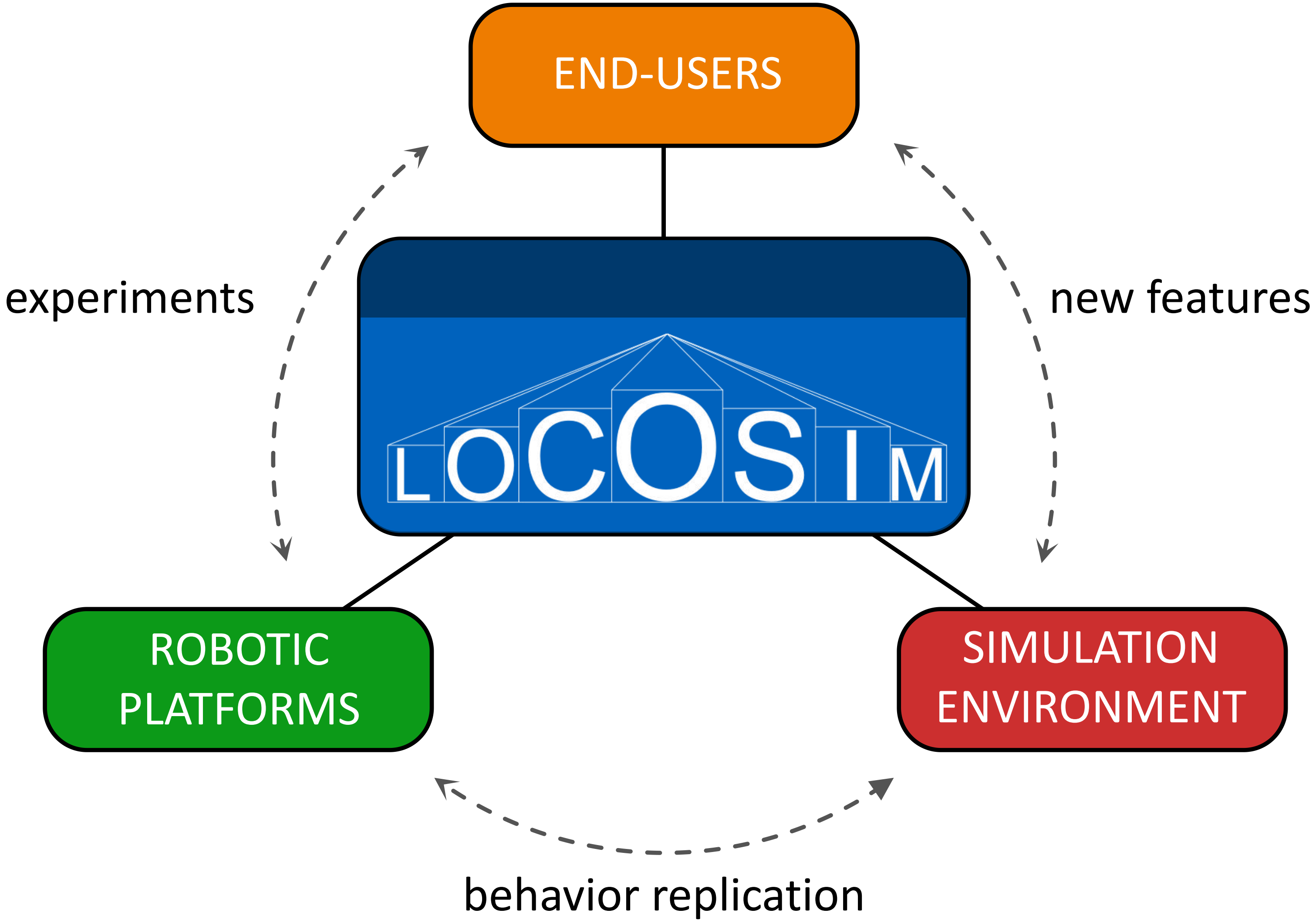






Critical requirements robotics framework




END-USERS
experiments
new features
LOCOSIM
ROBOTIC PLATFORMS
SIMULATION ENVIRONMENT
behavior replication

END-USERS

ROBOTICS FRAMEWORK

+ Generality

SW free from limitations and restrictions

ROBOTIC PLATFORMS

SIMULATION ENVIRONMENT

END-USERS

ROBOTICS FRAMEWORK

+ Generality

+ Modularity

Provide separate building blocks, each with tools for debug and test

ROBOTIC PLATFORMS

SIMULATION ENVIRONMENT

END-USERS

ROBOTICS FRAMEWORK

+ Generality
+ Modularity

+ Reusability

Reassemble codes for new functionalities with minor or no modifications

ROBOTIC PLATFORMS

SIMULATION ENVIRONMENT

END-USERS

ROBOTICS FRAMEWORK

+ Generality
+ Modularity
+ Reusability

+ Extensibility

Foresight to support evolutions of HW and SW

ROBOTIC PLATFORMS

SIMULATION ENVIRONMENT

END-USERS

ROBOTICS FRAMEWORK

+ Generality
+ Modularity
+ Reusability
+ Extensibility

+ Rapid Prototyping vs. Performances

Reduce coding and satisfy requirements for real robots

ROBOTIC PLATFORMS

SIMULATION ENVIRONMENT

END-USERS

ROBOTICS FRAMEWORK

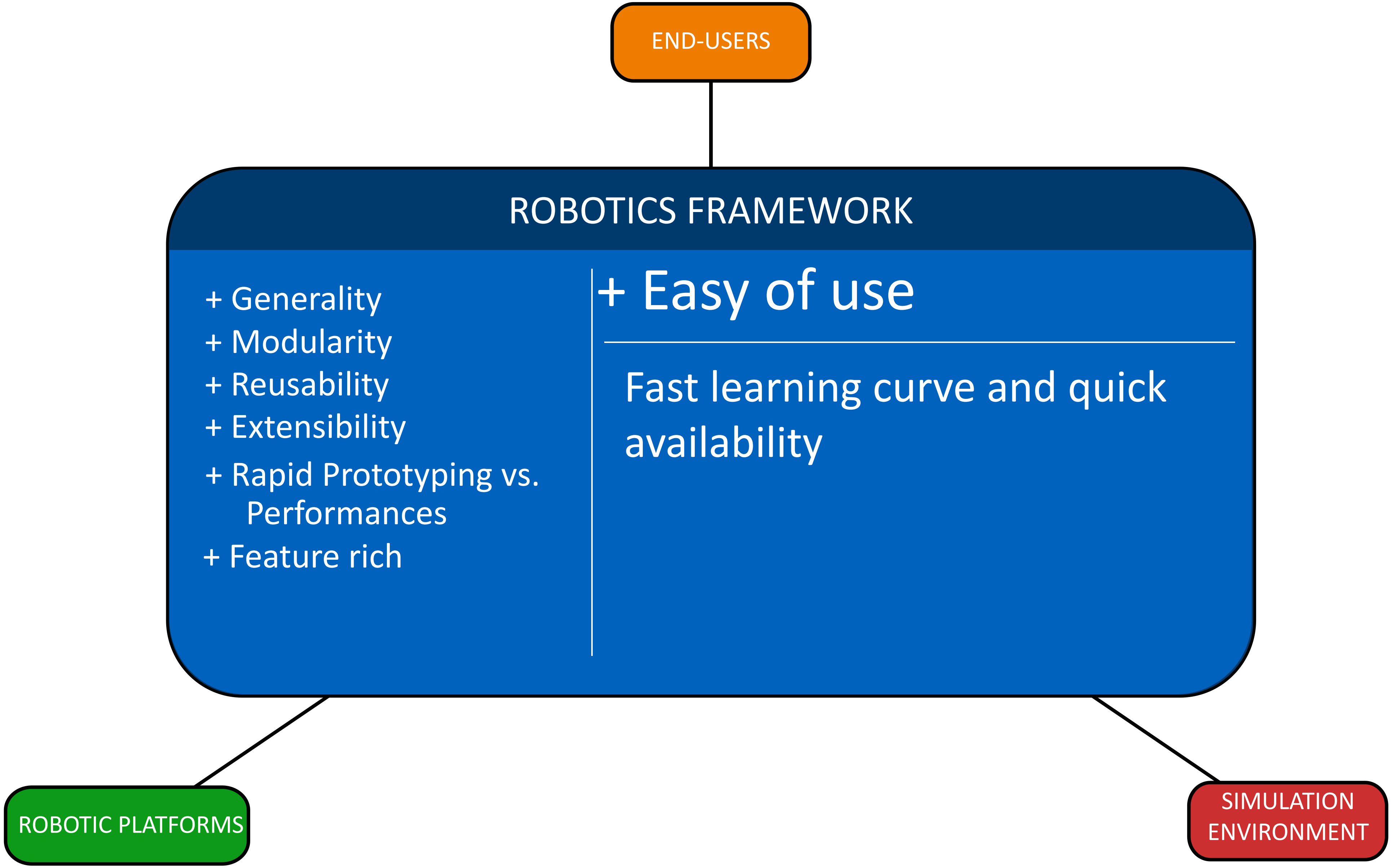
+ Generality
+ Modularity
+ Reusability
+ Extensibility
+ Rapid Prototyping vs. Performances

+ Feature-rich

Implement all basic functionalities for robotics

ROBOTIC PLATFORMS

SIMULATION ENVIRONMENT


END-USERS
ROBOTICS FRAMEWORK
+ Generality
+ Modularity
+ Reusability
+ Extensibility
+ Rapid Prototyping vs. Performances
+ Feature rich
+ Easy of use
Fast learning curve and quick availability
ROBOTIC PLATFORMS
SIMULATION ENVIRONMENT

Locosim Folder Structure

- robot_control @ 2260162
  - Lab experiments
  - Base classes for **fixed** and **floating-base** robots
  - Basic components (visualization, logging, etc)
- robot_descriptions
  - Repositories of the **xacro** descriptions for the supported robots
- robot_hardware_interfaces
  - Repositories of the hardware interfaces to communicate with the real robots (i.e. drivers, etc.)
- robot_urdf
- ros_impedance_controller @ e5b
  - Ros node that implements the low level PD + FFWD controller (Written in C++)
- .gitmodules
- Readme.md

- Robot_control

- base_controllers
- lab_exercises
- optimization
- rviz

base_controllers:

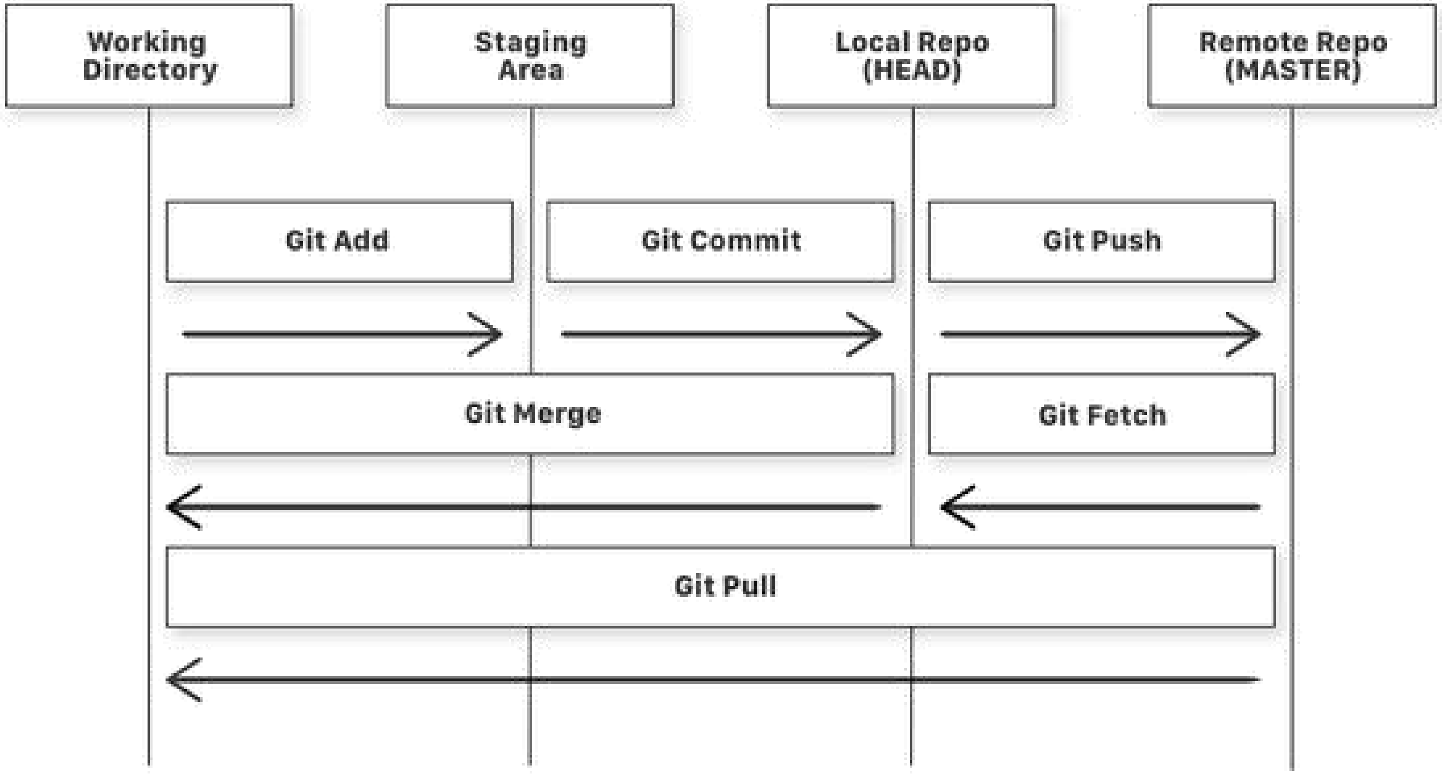
- components
  - Inverse kinematics
  - Controller Manager
  - Gripper Manager
  - Admittance Controller
  - Obstacle avoidance
  - Whole Body Controller
- utils
  - Robot Wrapper
  - Math utils
  - PID manager
  - Visual features (plot arrows, etc)
- __init__.py
- base_controller.py
- base_controller_fixed.py
- climbingrobot_controller.py

Git versioning system

- Git is a powerful **distributed version control system** that records changes to a file or set of files over time so that you can recall specific versions later (Ideally, you can place any file in the computer on version control)
- A Version Control System (VCS) allows you to revert specific files (or the entire project) back to a previous state, keep track of the changes made over time (history), see who last modified something that might be causing a problem, who introduced an issue and when.
- Git was originally authored by Linus Torvalds in 2005 for development of the Linux kernel
- Fundamental tool for coordinating work among programmers who are collaboratively developing source code, becaue it helps you synchronise code between multiple people (avoiding code replication)
- Git is a di**stributed** VCS because it does not necessarily rely on a central server to store all the versions of a project's files. Instead, every user "clones" a copy of a repository (locally). This means he has the full history of the project on his own hard drive.

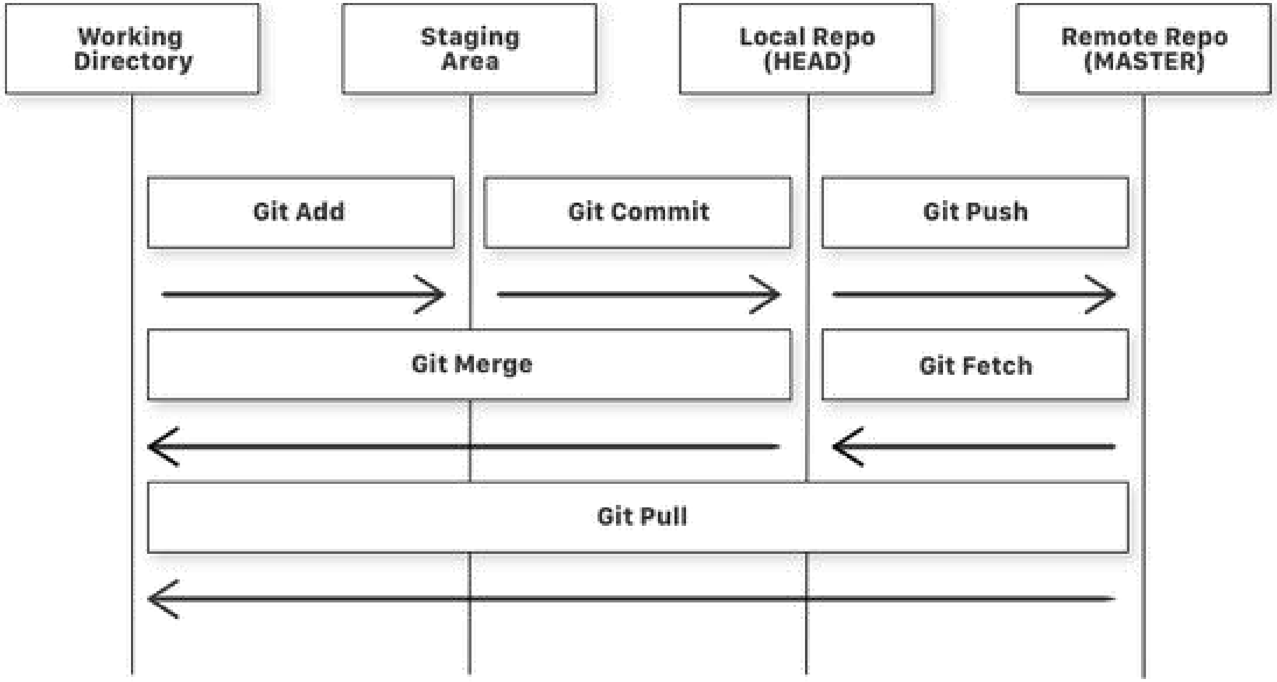
Git workflow

- A repository is nothing but a collection of source code.
- There are four fundamental elements in the Git Workflow.



Git states

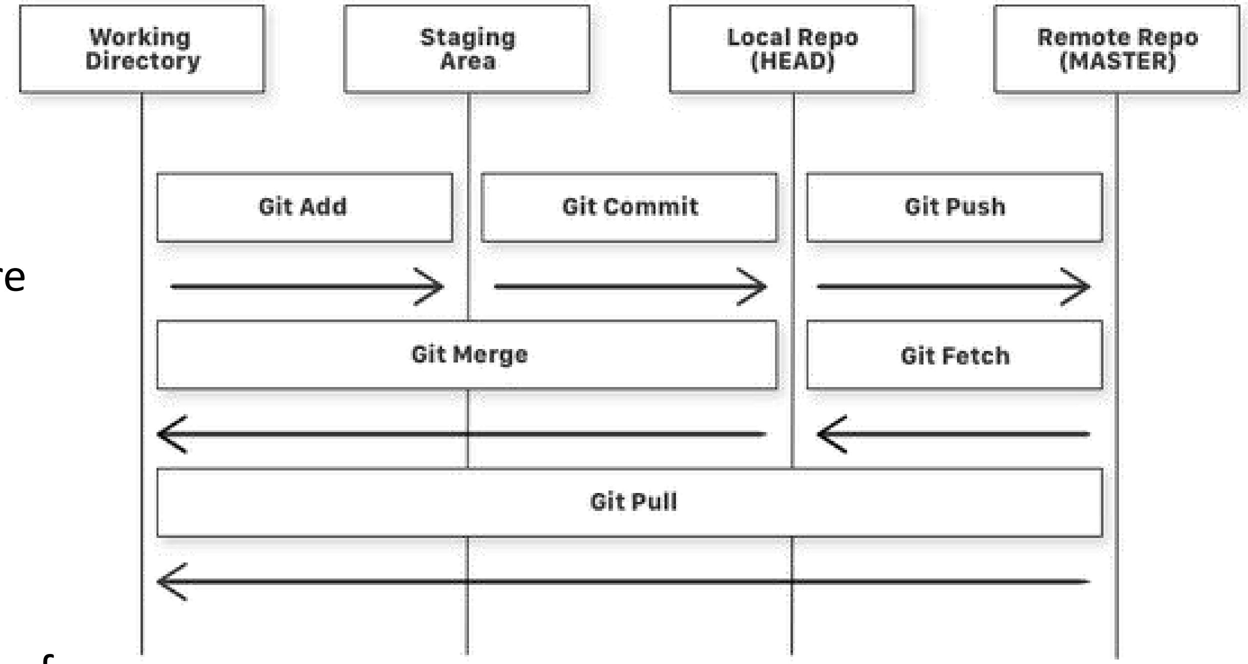
- If you consider a file in your Working Directory, it can be in three possible states:
  - **It can be modified.** Which means the files with the updated changes are not yet stored in the local repository.
  - **It can be staged**. Which means the files with the updated changes are **marked** to be committed to the local repository but not yet committed.
  - **It can be committed.** Which means that the changes you made to your file are safely stored in the local repository.



- git **add** is a command used to add a file that is in the working directory to the **staging** area.

Git Commands

- **git commit** is a command used to **store** the changes that are staged into the local repository.
- **git push** is a command used to propagate the commited changes in the local repository to the **remote** repository. So in the remote repository, all files and changes will be visible to anyone with access to the remote repository.
- **git fetch** is a command used to aknowledge changes from the remote repository to the local repository (but not store them into the working directory, yet)
- **git pull** is command used to get changes from the remote repository directly into the working directory. It is automatically doing a git fetch as first step.
- **git merge** is a command used to merge a different version of the repository (e.g. a branch) into the working directory.



Git submodules

- Locosim has a complex structure with many submodules
- Each submodule is a repository of code that implements a logically separated features (e.g. kinematic description of a robot, controller, planner, etc.)
- you can recognize a submodule by the presence of a @ symbol in GitHub. Each submodule is a repository on its own.
- The “mother” repository (i.e. Locosim) just points to the specific commits in each submodule that are compatible (e.g. represent a certain working state).
- The use of submodules is quite advanced and it really makes the difference in big projects with many components evolving separately
- In this course we just expect that you take care of your own repository for code development

Keep track of your own work

- To install git:

```
>sudo apt install git
```

- Useful tools are:

```
>sudo apt install git-cola
```

```
>sudo apt install git-g
```

- Only the first time you use git, tell Git who you are.

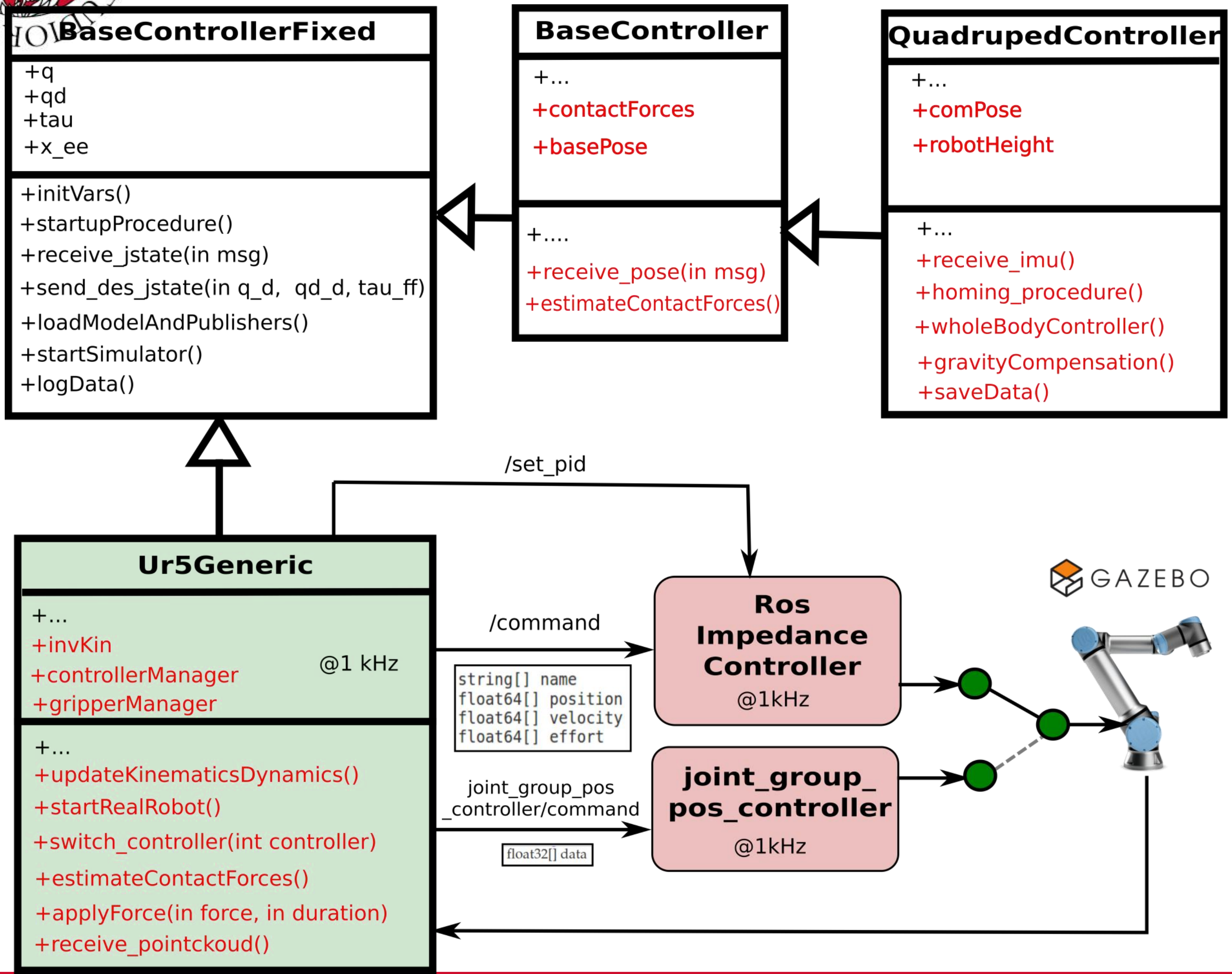
```
>git config --global user.name "YOUR_USERNAME"
>git config --global user.email "im_a_clever_student@musk.com"
```

- before starting a new work we suggest to create a new local branch on the submodule repository you want to modify by typing the command:

```
>git checkout -b my_new_branch
```

- To periodically save your changes you can use the **git-cola** gui, and “stage” the changes and associate a commit to them, keeping a snapshot of your work
- From now on, each commit will be stored in your local branch called **my_new_branch**
- You can review the history of your commits using **gitg** in the repository folder

Locosim Software architecture



- the **Ur5Generic** planner inherits from the **BaseControllerFixed** base class
- **ros_control** is a set of packages to make controllers generic to all robots
- The low level controllers (ros_impedance_controller and joint_group_pos_controller) are based on **ros_control** package and receive joint commands from Python and send them to Gazebo
- a **/set_pid** service call allows to set the PD gains

C++

Python

Robot models and URDF

A simulator needs a (mathematical) model to predict the robot's behaviour, based on the laws that govern the motion of the mechanical structure

- Nowadays, a commonly used format to describe robots is the so-called Unified Robot Description Format (URDF)

- The **Unified Robot Description File (URDF)** is a domain specific file format based on XML that describes:
  - The body layout of the robot (kinematic and collisions surfaces)
  - Visual appearance
  - Information about joint position and velocity limits
  - It can also include sensor models (e.g. cameras, LiDar) through plugins

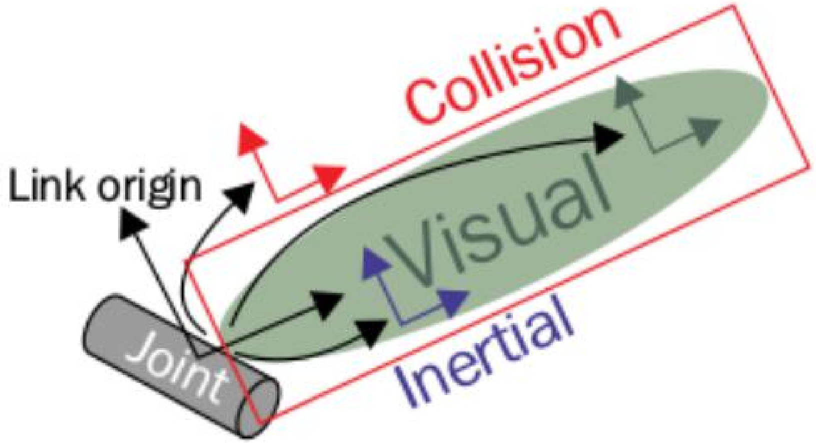
- To improve modularity, reusability we employ the macro language for XML called **XACRO** that allows to construct more readable XML files adding **parameters**

- XACRO allows to define Constants , macros, and perform Simple Math and allows to simplify a lot the URDF files

Assemblying the URDF: the link tag

- The **link** tag represents a single link of a robot. Using this tag, we can model a robot link and its dynamic properties. The syntax is as follows:

```
<link name="<name of the link>">
        <inertial>...........</inertial>
        <visual> ............</visual>
        <collision>..........</collision>
</link>
```



- The **visual** tag represents the visual shape of the link

- The area surrounding the real link is the **Collision** section. This encapsulates the real link to detect collision before hitting the real link.

- The **inertial** tag defines the mass, the location of the Center of Mass and the inertia tensor of the link (about the CoM).

From CAD to URDF: example, a quadruped upperleg link

VOLUME = 1.2935215e+06 MM^3
SURFACE AREA = 7.0070859e+05 MM^2
AVERAGE DENSITY = 3.8261849e-06 KILOGRAM / MM^3
MASS = 4.9492523e+00 KILOGRAM

CENTER OF GRAVITY with respect to HFE_REF coordinate frame:
X Y Z 1.4429642e+02 -2.0705973e+00 -5.0939522e+00 MM

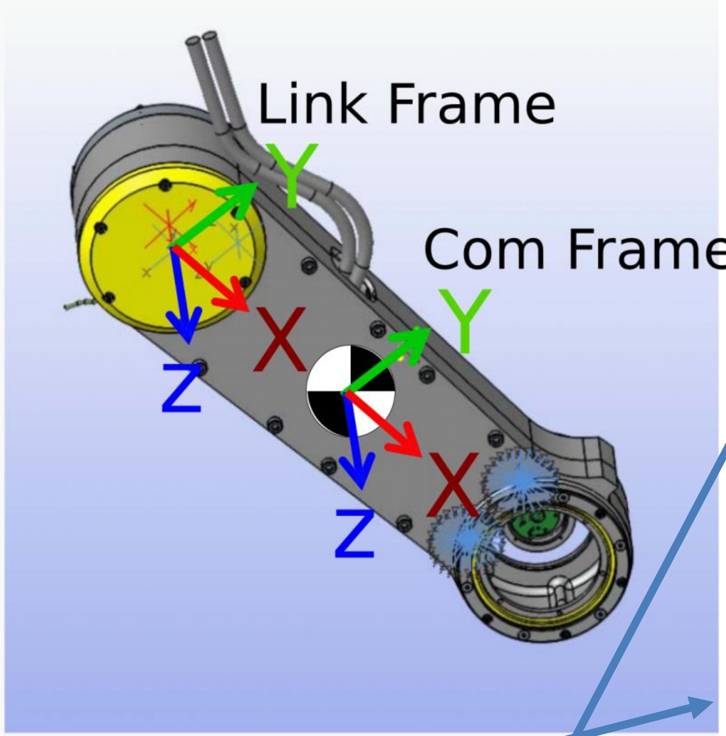
INERTIA with respect to HFE_REF coordinate frame: (KILOGRAM * MM^2)

INERTIA TENSOR:
Ixx Ixy Ixz 1.0195762e+04 2.3200719e+03 1.4058423e+02
Iyx Iyy Iyz 2.3200719e+03 2.0418300e+05 -3.3927652e+01
Izx Izy Izz 1.4058423e+02 -3.3927652e+01 2.0753205e+05

INERTIA at CENTER OF GRAVITY with respect to HFE_REF coordinate frame:
(KILOGRAM * MM^2)

INERTIA TENSOR:
Ixx Ixy Ixz 1.0046118e+04 8.4133545e+02 -3.4973095e+03
Iyx Iyy Iyz 8.4133545e+02 1.0100393e+05 1.8274704e+01
Izx Izy Izz -3.4973095e+03 1.8274704e+01 1.0446019e+05

PRINCIPAL MOMENTS OF INERTIA: (KILOGRAM * MM^2)
I1 I2 I3 9.9089747e+03 1.0101166e+05 1.0458961e+05

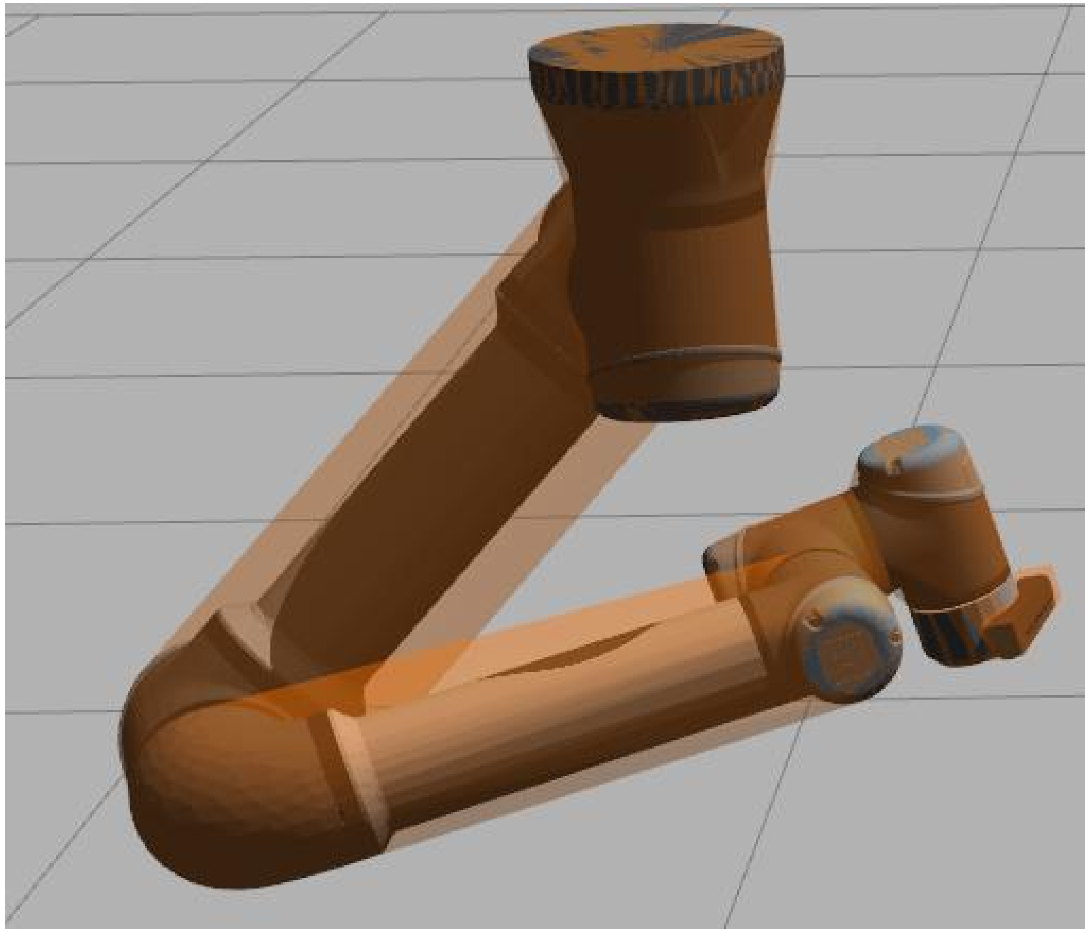


- These parameters are usually obtained by CAD. SI units should be used
- The tensor should be a positive definite matrix!

Com position w.r.t. link frame

```
<inertial>
        <origin xyz=“ 0.144 -0.002 -0.005 "/>
        <mass value=“4.949"/>
        <inertia ixx=“0.01"  iyy=“0.204“ izz=“0.207“ ixy=“0.00008” ixz=“-0.0035” iyz=“0.00002"/>
</inertial>
```

Collisions for the UR5 robot



- The collision tags define their shape the same way the visual element does, with a **geometry** tag
- They can be set equal to the visual mesh. However, doing collision detection for two meshes is a lot more computational complex than for two simple geometries. Hence, you may want to replace the meshes with **simpler** geometries in the collision element.
- A complex 3D mesh can be simplified using the **meshlab** tool. **Tutorial**: www.youtube.com/watch?v=mK1n35gnpg4
- This is useful to limit the maximum number of contacts between two entities, such as face-to-face collisions, (potential sacrifice to the accuracy)

Assembling the URDF: the joint tag

- The joint tag represents a robot joint. We should specify the type of joint revolute, prismatic, floating, fixed, continuous).

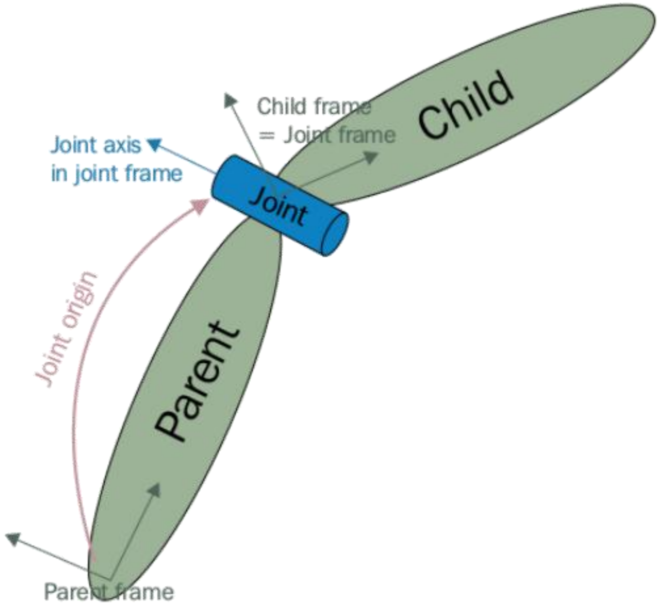
- A joint is formed between two links; the first is called the **Parent** link and the second is the **Child** link:

```
<joint name="<name of the joint>" type"<type of the joint>"
        <parent link="link1"/>
        <child link="link2"/>
        <origin xyz="..." rpy="..."/>
        <axis xyz="0 0 1"/>
        <limit effort .... />
</joint>
```

Rigid transform

Joint axis

Joint limits (torque, position, velocity)



- We can specify the kinematics of the robot by setting the location of the joint frame with respect to the **supporting** link with the tag **origin** that represents the rigid transform with respect to the supporting link frame

- In this picture the frame supporting the child link is coincident with the joint frame so it means that the joint variable is 0

Assembling the URDF: the transmission tag

- The transmission element is an extension to the URDF robot description model that is used to describe the relationship between an actuator and a joint.
- This allows one to model concepts such as gear ratios and parallel linkages.
- Currently, **only** the ros_control project uses this transmission elements.

```
<transmission name="shoulder_lift_trans">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="shoulder_lift_joint">
        <hardwareInterface>PositionJointInterface</hardwareInterfa ce>
    </joint>

   <actuator name="shoulder_lift_motor">
        <mechanicalReduction>1</mechanicalReduction>
   </actuator>
</transmission>
```

Associated joint

Associated **hardware interface**: {PositionJointInterface, VelocityJointInterface, EffortJointInterface}

Visualizing a model described by the URDF in RVIZ

- Locosim already provides a dedicated function to publish the state of the joints during runtime and make them available to the visualizer node called RVIZ.
- However, to debug kinematics, it is useful to use the **joint_state_publisher_gui** to set the joint states
- For each supported robot you can run the following command using the terminal

>roslaunch ur_description rviz.launch

- this is a **launch** file that runs multiple nodes with only one command

```
<?xml version="1.0"?>
<launch>
	<include file="$(find ur_description)/launch/upload.launch"/>
	<node name="joint_state_publisher_gui" pkg="joint_state_publisher_gui" type="joint_state_publisher_gui" />
	<node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
	<node name="rviz" pkg="rviz" type="rviz" args="-d $(find ur_description)/rviz/conf.rviz"/>
</launch>
```

Loads the URDF description in the parameter server

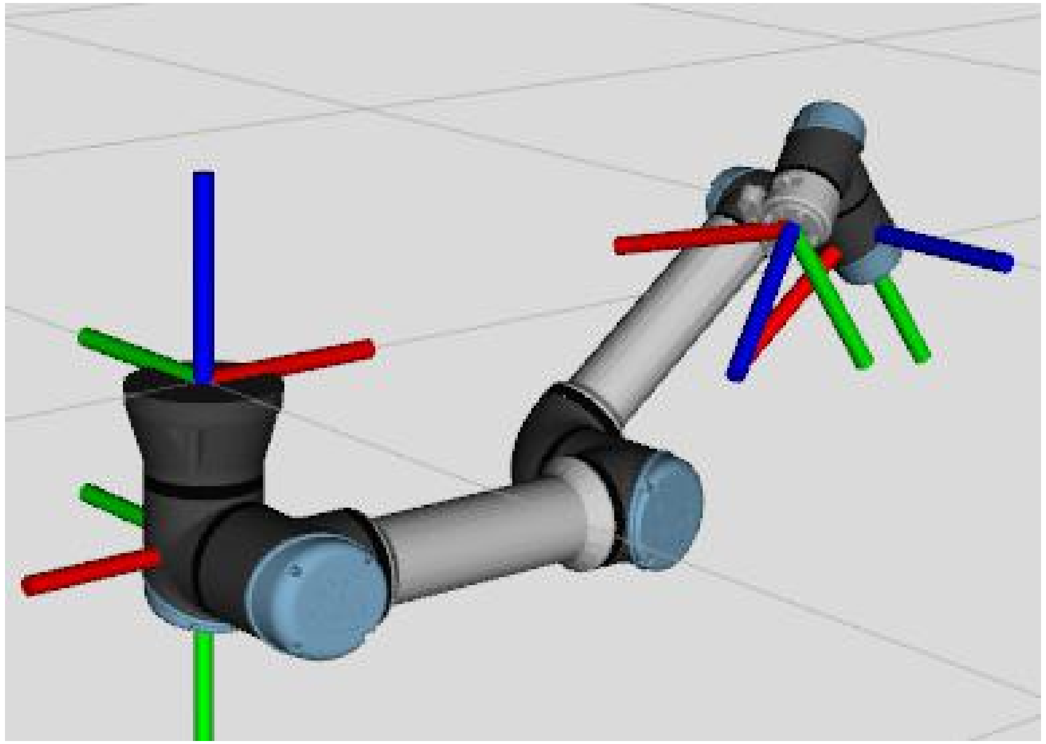
Loads the GUI to set joint positions and publish the **/joint_states** topic

Computes the TFs (i.e. hom. transforms) from the **/joint_states** topic and publish them in the **/tf** topic

Launches RVIZ

Visualizing a model described by a URDF in RVIZ

RVIZ visualizer

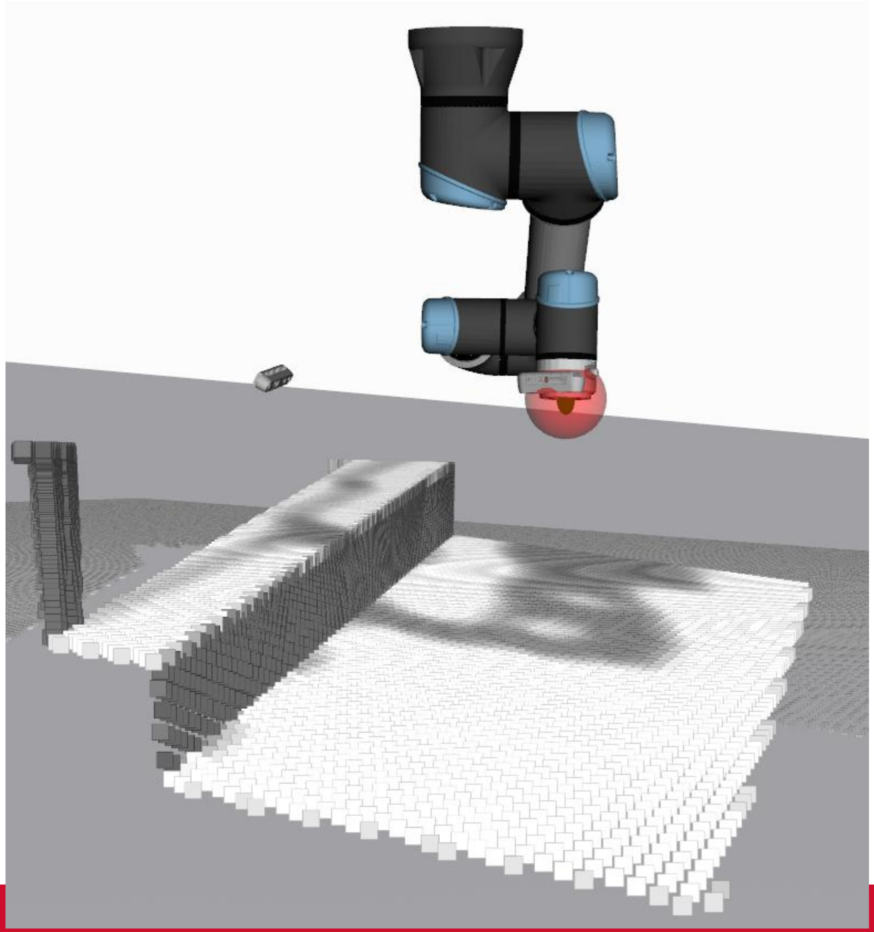


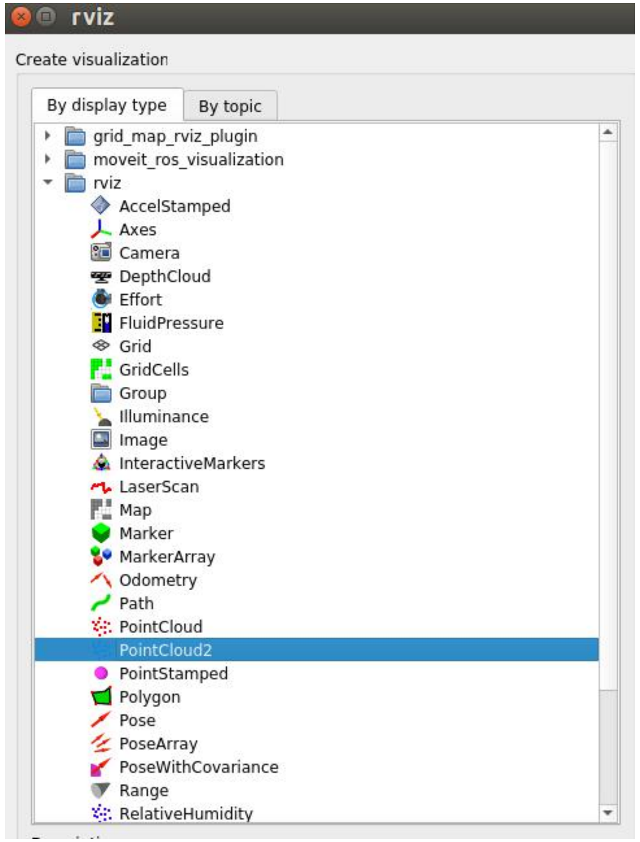
Joint state publisher GUI

- You can play around with the value of the joint angular positions, using the slider input in the GUI
- Verify the value of the assigned joint positions with: **>rostopic echo /joint_states**

Attach a Realsense stereo-camera to the end-effector

- to be able to see the acquired point-cloud you should add the **pointcloud2** display-type plugin in RVIZ and select the appropriate topic





- Note the point-cloud is produced in the camera frame, but rviz allows you to show it in the frame you specified in “global options”(default world)

Include objects in a custom world file

- in the ros_impedance_controller_XX.launch file, where the **gazebo_ros** package is loaded, it is possible to specify a custom world file different than the default (called empty.world):

```
<include file="$(find gazebo_ros)/launch/empty_world.launch" >
          <arg name="world_name" value="$(find your_package)/your_world.world”/>
        <arg name="paused" default="false" />
        <arg name="use_sim_time" default="true" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="headless" default="false" />
        <arg name="debug" default="false" />
        <arg name="verbose" value="true" />
</include>
```

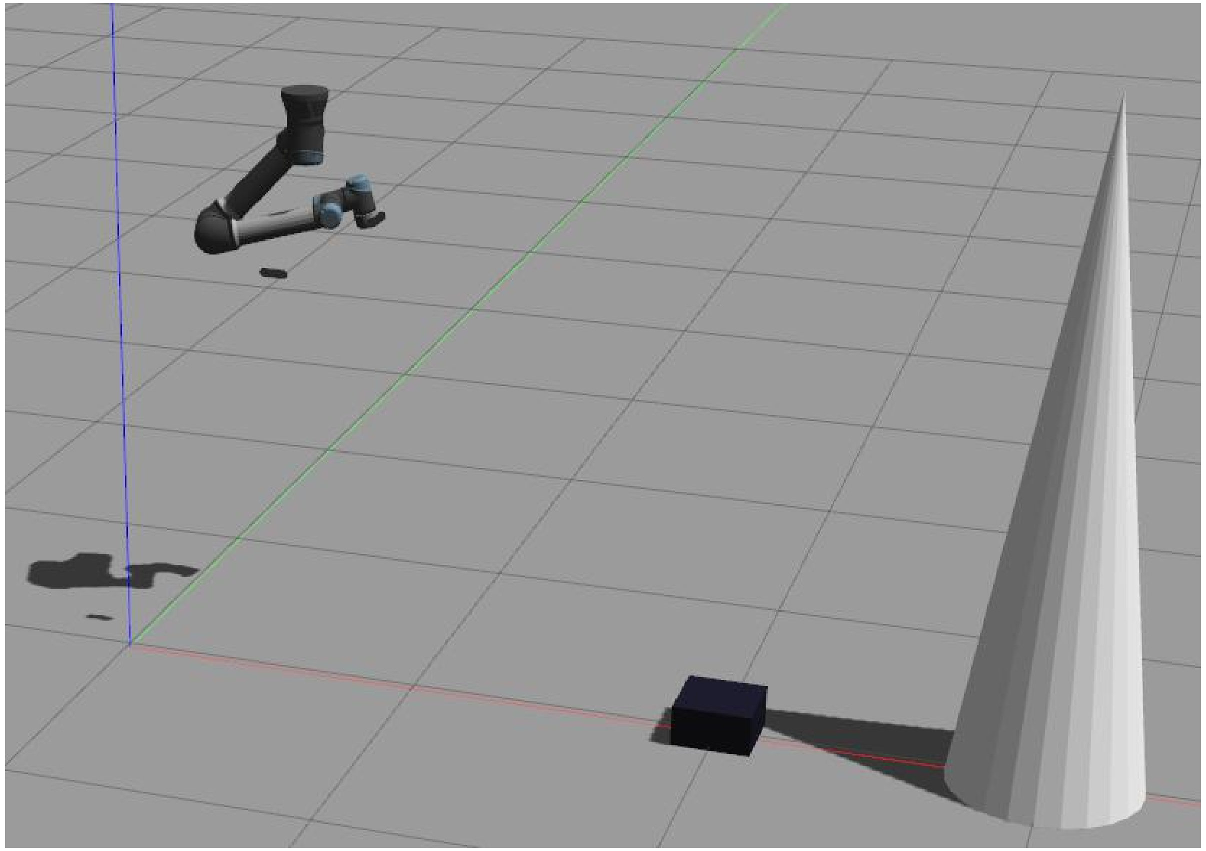
- in a world file you can programmatically include models of objects in specific locations of the simulation environment

- these models can be part of the Gazebo library or custom made bythe user (e.g. from CAD)

- alternatively, the objects can be added by the left-panel and manually dragged to a desired location (not precise). Only the models in the folders specified in $GAZEBO_MODEL_PATH are listed in the left-panel

Include objects in a world file

- the models should be defined in **sdf** format which is slightly different from URDF, and is the format that Gazebo wants in input (URDF is also automatically translated to sdf)



- let's add a 25 cm side box at position [2 0 0] m and a custom cone at position [6 0 0]m

- because the cone is not part of the default supported shapes (boxes, spheres, and cylinders) we should draw it with a CAD software and export it in **STL** format

sdf model of a box

```
<?xml version='1.0'?>
<sdf version="1.4">

  <model name="box">
    <pose>0. 0. 0. 0. 0. 0.</pose>
    <static>true</static>
    <link name="link">
      <inertial>
          <mass>1.0</mass>
          <inertia>  <ixx>0.08</ixx> <ixy>0.0</ixy> <ixz>0.0</ixz>
                         <iyy>0.08</iyy> <iyz>0.0</iyz> <izz>0.08</izz>
          </inertia>
      </inertial>
      <collision name="collision">
        <geometry>
          <box>
            <size>0.25 0.25 0.25</size>
          </box>
        </geometry>
      </collision>
      <visual name="visual">
        <geometry>
          <box>
            <size>0.25 0.25 0.25</size>
          </box>
        </geometry>
      </visual>
    </link>
  </model>
</sdf>
```

- change this if you want to shift the reference frame of the object from the center (e.g. put in one of the corners)
- parameter **“static”** makes the model immovable. Set to false if you want your model to be movable.
- replace this tags with the CAD mesh for the custom shape **cone** object

```
<geometry>
    <mesh>
        <uri>model://cone/meshes/cone.stl</uri>
        <scale>0.001 0.001 0.001</scale>
    </mesh>
</geometry>
```

- tipically CAD softwares output the models in mm so you might need to use the **scale** tag

Putting everything together...palopoli.world

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="default">
    <physics type='ode'>
      <gravity>0 0 -9.81</gravity>
      <max_step_size>0.001</max_step_size>
      <real_time_update_rate>1000</real_time_update_rate>
    </physics>

    <include>
      <uri>model://sun</uri>
    </include>
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <include>
      <name>my_box</name>
      <uri>model://box</uri>
      <pose> 2. 0. 0.  0. 0. 0.</pose>
    </include>

    <include>
      <name>my_cone</name>
      <uri>model://cone</uri>
      <pose>6. 0. 0.  0. 0. 0.</pose>
    </include>

  </world>
</sdf>
```

- type of the dynamics engine: ode (default), bullet, dart
- frequency at which the simulation time steps are advanced.
- location of the center of the box
- location of the main frame of the cone (depends on how did you define it in CAD)
- NOTE: Since these files are located inside a package, remember to compile to be sure they are installed in the devel space which is in the PYTHONPATH

Create a new floating base robot (optional)

- floating base robots have the base that is underactuated, here some examples:

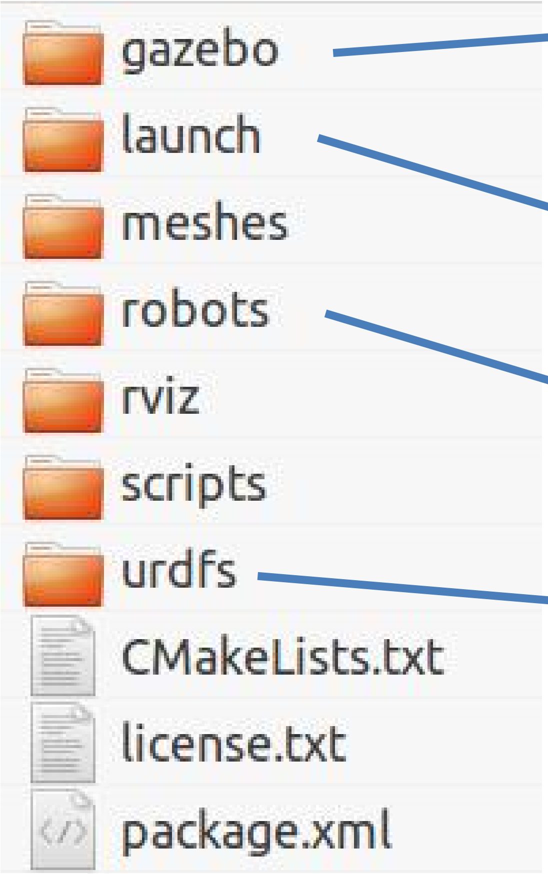
quadrupeds

snake robots

mobile robots

Create a new floating base robot

- create a new ros package: e.g. called **myrobot**_description, the folder structure to work with locosim is:



- **gazebo.urdf.xacro**: launches ros_control package and the ground_truth publisher that publishes the position/orientation of the robot truk (needed only for floating-base robots)

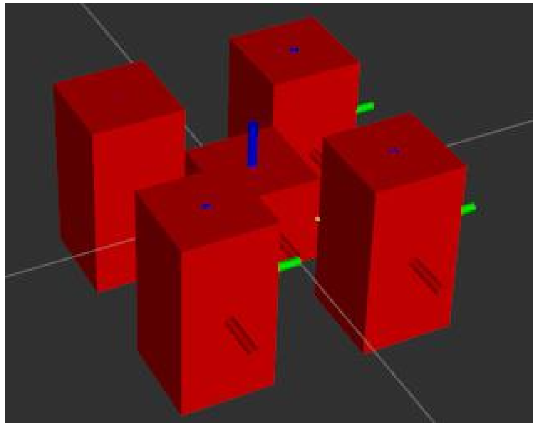
- **upload.launch**: process the xacro generating the URDF and loading into the parameter server

- **myrobot.urdf.xacro**: xacro of the whole robot, is like the "assembly", it imports other urdfs (e.g. legs) that are in the urdfs folder, specifying their location

- **myleg.xacro**: xacro of the sub-assembly of the robot (e.g a leg)
  - **- myleg.transmission.xacro**: trasmissions for each joint (both active and passive ones)
  - - frictional properties of links
  - - contact sensors

Create a quadruped robot

- We want to create a quadruped with a **base link** and 4 legs, with a single joint per leg



- we first include the xacro definition of the a macro called “leg”

- each leg is instantiated with the macro “leg” by specifying the location

```
<?xml version="1.0" ?>
<robot name="myrobot" xmlns:xacro="http://ros.org/wiki/xacro">
    <xacro:include filename="$(find myrobot_description)/gazebo/gazebo.urdf.xacro"/>
    <xacro:include filename="$(find myrobot_description)/urdfs/myleg.xacro"/>
    <xacro:arg name="robot_name" default="myrobot"/>

    <link name="base_link" />
        <inertial>
            <origin xyz="0.0 0.0 0.0"/>
            <mass value="15" />
            <inertia  ixx="0.1" ixy="0.0"  ixz="0.0"  iyy="0.1"  iyz="0.0"  izz="0.1" />
        </inertial>
        <visual>
            <origin xyz="0 0.0 0.0" rpy="0 0 0"/>
            <geometry>
                <box size="0.1 0.1 0.1"/>
            </geometry>
        </visual>
        <collision>
            <origin  xyz="0 0.0 0.0 " rpy="0 0 0"/>
            <geometry>
                <box size="0.1 0.1 0.1"/>
            </geometry>
        </collision>
    </link>
  <xacro:leg
        name="lf"
        parent="base_link"
        <origin xyz="0.1 0.1 0.0" rpy="0 0 0"/>
  </xacro:leg>
  <!-- same for other LEGS -->
</robot>
```

Create a quadruped robot

- the macro leg includes the type of transmission for the only joint (**shoulder_pan_joint)**
- we chose an **EffortJointInterface** transmission to be able to implement both position and torque control

```
<xacro:macro name="leg" params="name parent  *origin ">
    <joint name="${name}_shoulder_pan_joint" type="revolute">
            <parent link="${parent}"/>
            <child link="${name}_shoulder_link"/>
            <axis xyz="0 0 1"/>
            <xacro:insert_block name="origin"/>
            <limit effort="50" velocity="10.0" lower="-3.14" upper="3.14" />
    </joint>

    <link name="${name}_shoulder_link">
        <inertial>
            <origin xyz="0.0 0.0 0.0"/>
            <mass value="0.05" />
            <inertia  ixx="0.001" ixy="0.0"  ixz="0.0"  iyy="0.001"  iyz="0.0"  izz="0.001" />
        </inertial>
        <visual>
                <origin xyz="0 0.0 0.0" rpy="0 0 0"/>
                <geometry>
                    <box size="0.1 0.1 0.2"/>
                </geometry>
                </visual>
                <collision>
                    <origin  xyz="0 0.0 0.0 " rpy="0 0 0"/>
                    <geometry>
                            <box size="0.1 0.1 0.2"/>
                    </geometry>
                </collision>
            </link>
    <xacro:leg_transmission name="${name}"/>
</xacro:macro>
```

Create a quadruped robot

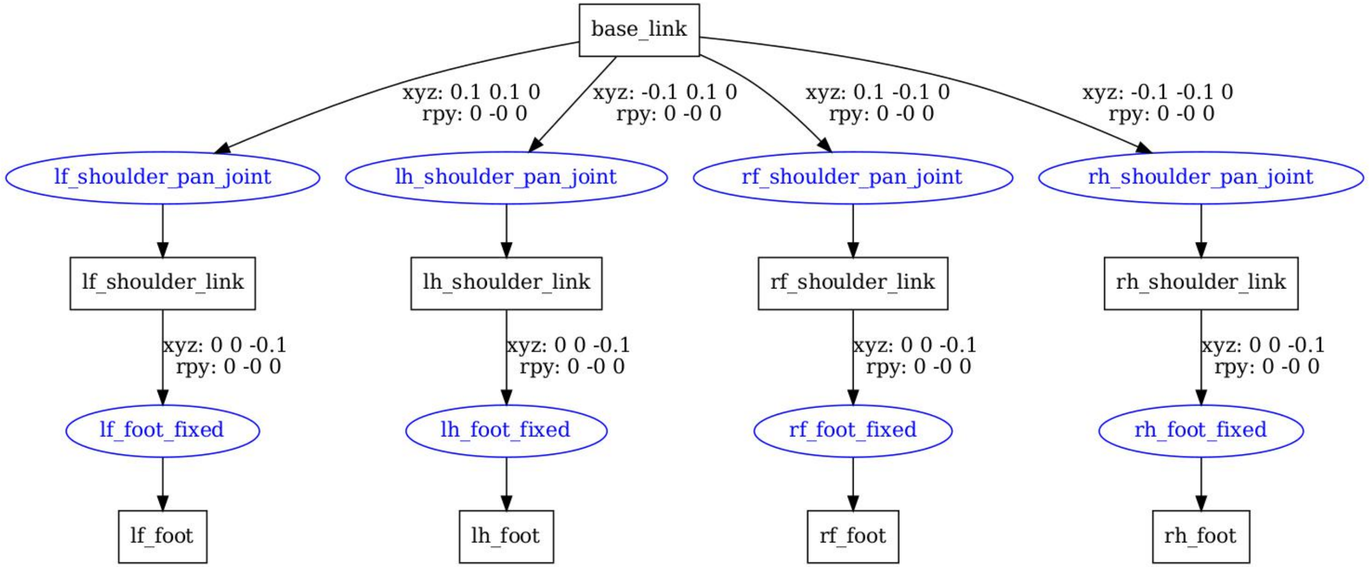
- it miss to create the YAML config file for the PD gains in myrobot_description/config

```
ros_impedance_controller:
  type: ros_impedance_controller/Controller
  joints:
    - lf_shoulder_pan_joint
    - rf_shoulder_pan_joint
    - lh_shoulder_pan_joint
    - rh_shoulder_pan_joint

  gains:
    lf_shoulder_pan_joint: {p: 10.0, i: 0.0, d: 1.0}
    rf_shoulder_pan_joint: {p: 10.0, i: 0.0, d: 1.0}
    lh_shoulder_pan_joint: {p: 10.0, i: 0.0, d: 1.0}
    rh_shoulder_pan_joint: {p: 10.0, i: 0.0, d: 1.0}

home:
    lf_shoulder_pan_joint: 0.
    rf_shoulder_pan_joint: 0.
    lh_shoulder_pan_joint: 0.
    rh_shoulder_pan_joint: 0.
```

- to simulate we can run the script **lab_palopoli/generic_simulator.py** setting the variable robotName = "myrobot"

Create a quadruped robot

- to check if the kinematics tree is correct you can build the kinematics graph



- you can obtain it first generating the urdf from the xacro

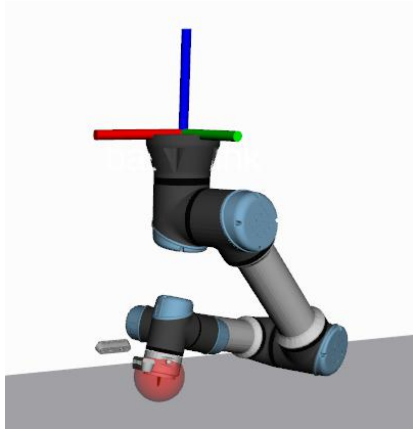
>rosrun xacro xacro myrobot_package_folder/robots/myrobot.urdf.xacro -o **myrobot.urdf**

- then calling the following command (you need to install liburdfdom-tools package) that will create a myrobot.pdf

>urdf_to_graphiz **myrobot.urdf**

Create a fixed base robot

- fixed base robots have all the joints actuated and the base that does not move (e.g. industrial manipulators)



- the difference in creating a fixed base robot is that you should name the root link as “**world”** without inertia

```
<link name="world"/>
```

- then you create a fixed link between the base_link and the root world link

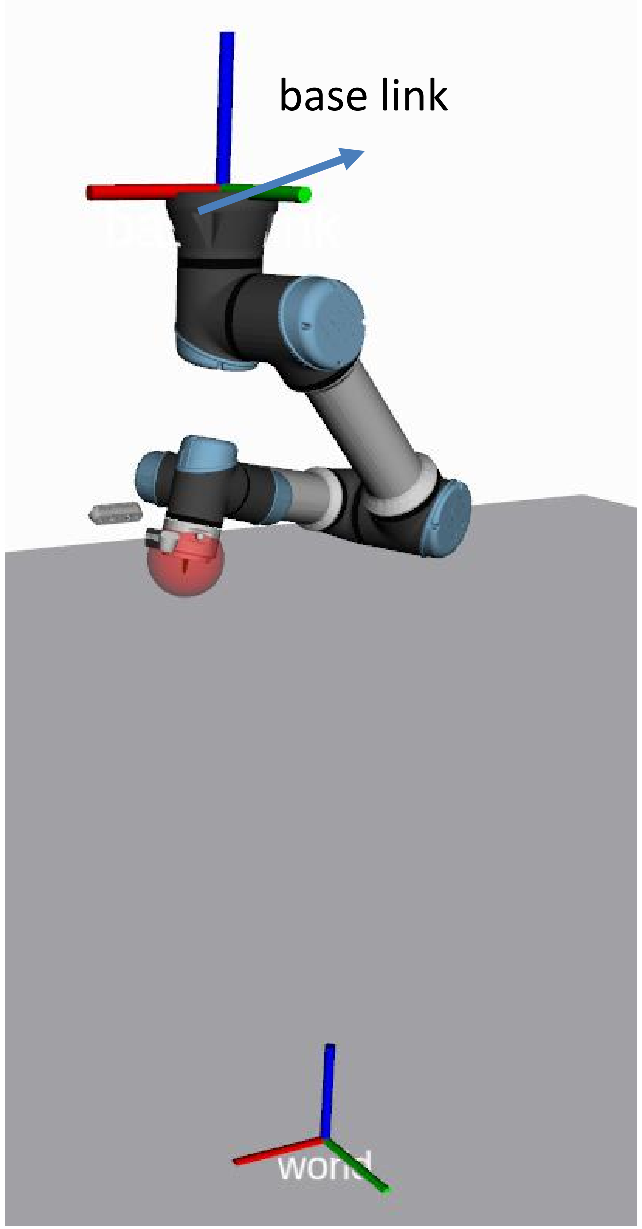
```
<joint name="world_to_base_link=" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
< /joint>
```

Spawn the robot in a custom location

- you can spawn both the fixed base and floating base robots in a different location (e.g [0.5, 0.5, 2] with respect to the world frame by specifying it in the file where you launch the gazebo_ros package (i.e. ros_impedance_controller_XX.launch):

```
<node name="spawn_gazebo_model" pkg="gazebo_ros" type="spawn_model"
args="-urdf -param robot_description -model myrobot -x 0.5 -y 0.5 -z 2 "/>
```

- in Locosim you can just set the spawn_x, spawn_y, spawn_z variables in params.py



Controlling the robot

- Locosim contains a PID Manager class that manages a PID + feed-forward controller (implemented in C++ in the ros_impedance_controller class). The controller computes joint torques (to be given as inputs to Gazebo) that realise the following control action:

$$\tau = K_p(q^d - q) + K_d(\dot{q}^d - \dot{q}) + K_i \int (q^d - q)dt + \tau_{ffwd}$$

- The feed-forward term τ_{ffwd} allows us to provide additional torque (e.g. to compensate gravity)
- The set-points for the joints should be assigned to the array q^d
- The PID Manager class exposes a service call /set_pid that can be called directly by the terminal:

```
>rosservice call /set_pid
```

- As an alternative you can set the PIDs in the code by using the PID manager methods (setPDs, setPDjoint, etc.).
- Note that, since it provides a torque input (and not a position input) to Gazebo, the PID Manager will be active only if control_type = 'torque'

Set points

- Now let us try to set the points for the joints to follow a 0.1 rad, 0.5 Hz sinusoidal trajectory around q 0 , uncommenting the following lines in the code:

```
p.q_des = p.q_des_q0 + 0.1 * np.sin(2*np.pi*0.5*p.time)*np.ones(p.robot.na)
p.qd_des = 0.1 * 2 * np.pi * 0.5* np.cos(2 * np.pi * 0.5 * p.time)*np.ones(p.robot.na)
p.controller_manager.sendReference(p.q_des, p.qd_des, p.g)
```

- Note that we computed the derivative of the position set-point, to have a consistent set-point for the velocity qd_des
- We also added torques as feed-forward p.g to compensate gravity where p.robot.na is the number of actuators (in our case 6).
- Let’s start the simulation (with control_type= ’torque’) and see if we are publishing on the right topic
- we can inspect the published topic as follows:

```
>rostopic echo / command
```

Setting PID

- The joints are nicely tracking the references, the default values for the PD gains are set in the following parameters in the params.py:

```
’kp’: np.array([300., 300., 300.,30.,30.,1.])
’kd’: np.array([20.,20.,20.,5., 5., 0.5])
```

- Let’s try now to reduce the proportional gain to 10 and the derivative gain to 0 for the first joint ’shoulder_pan_joint’

- We can do it in the code: p.pid.setPDjoint(0, 10, 0, 0) where the first entry 0 is the index of the first joint.

- or by directly calling the service call \set_pid and filling the message as follows:

```
"data:
- joint_name: ’shoulder_pan_joint’
p_value: 10.0
i_value: 0.0
d_value: 0.0"
```

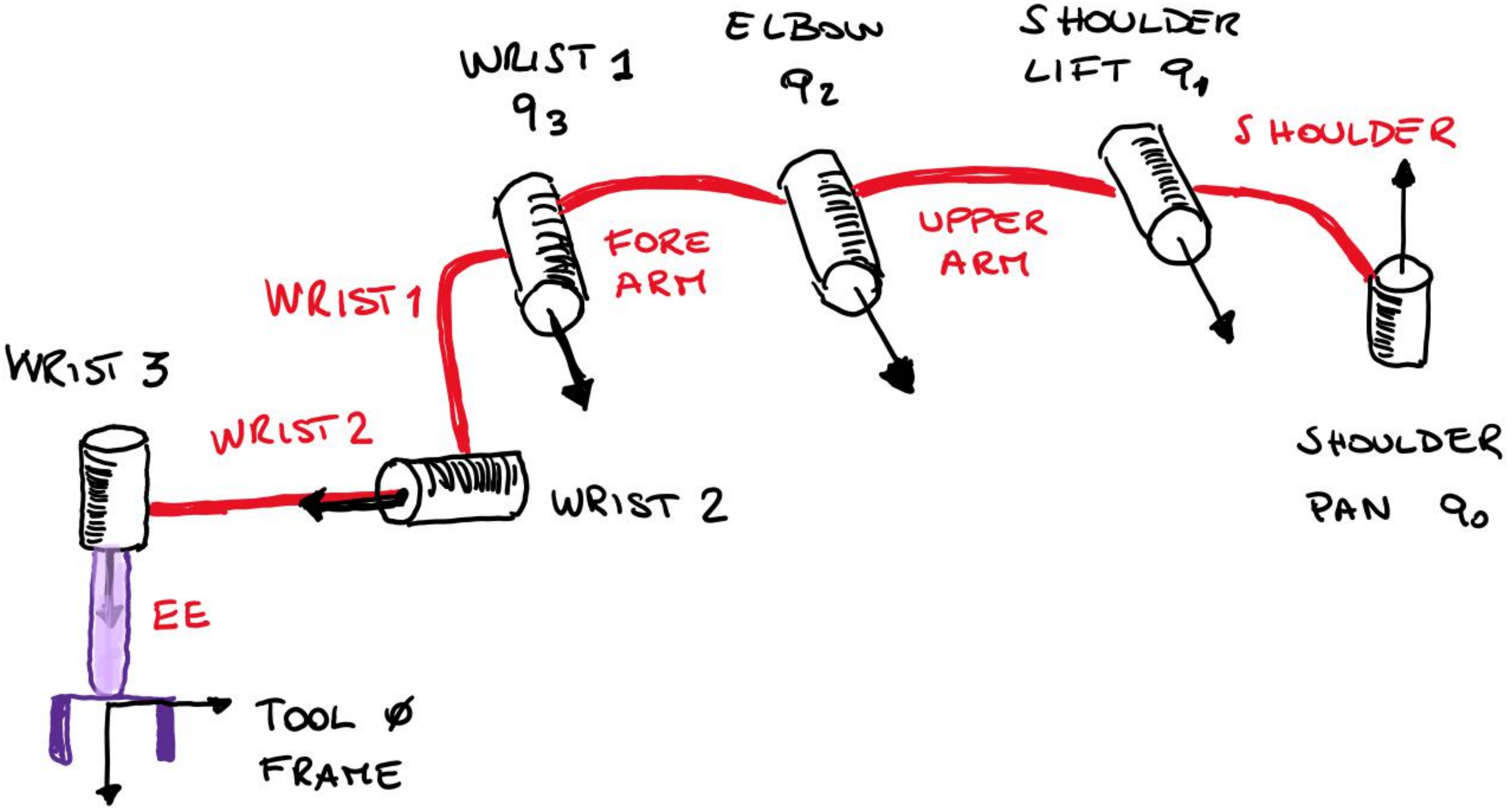
Setting PID

- You will see that that the ’shoulder_pan_joint’ joint will change its behaviour and start to overshoot

- Try to restore the original value and see what happens...

- if you press CTRL+C the code will exit from the infinite while loop and do a plot of the trajectories. This allows you to quantify whatever you have just seen in the simulation.

Exercise: setting a configuration for the UR5 joints with a publisher



Scketch of the UR5 kinematics

Create a custom joint state publisher (rospy)

```
#!/usr/bin/env python
import rospy as ros
import numpy as np
from std_msgs.msg import Float64MultiArray

class JointStatePublisher():
    def __init__(self):
        self.q_des =np.zeros(6)

    def send_des_jstate(self, q_des):
        msg = Float64MultiArray()
        msg.data = q_des
        self.pub_des_jstate.publish(msg)

def talker(p):
    ros.init_node('custom_joint_pub_node', anonymous=True)
    p.pub_des_jstate = ros.Publisher("/ur5/joint_group_pos_controller/command",
                                     Float64MultiArray)

    loop_rate = ros.Rate(1000.)  # 1000hz
    q_des0 = np.array([-0.3, -0.78, -2.56, -1.63, -1.57, -1.0])
    while not ros.is_shutdown():
        p.q_des = q_des0
        p.send_des_jstate(p.q_des)
        loop_rate.sleep()

if __name__ == '__main__':
    myPub = JointStatePublisher()
    try:
        talker(myPub)
    except ros.ROSInterruptException:
        pass
```

- Goal: we want to send the following joint state set-point to the robot: [-0.3, -0.78, -2.56, -1.63, -1.57, -1.0]
- let's define a Class called **JointStatePublisher** that does the job!
- the class contains only one method: **send_des_jstate()** that will publish the joints set-points on the topic **/ur5/joint_group_pos_controller/command** a message of type **Float64MultiArray**

float32[] data

- For simplicity let's not consider the gripper joints, hence set the parameter 'gripper_sim': False in the params.yaml. The gripper will be considered a rigid body

- Makes sure your script is executed as a Python script.
- Import of the necessary packages (rospy , numpy), the std_msgs.msg import is so that we can reuse the Float64MultiArray for publishing

Create a custom joint state publisher (rospy)

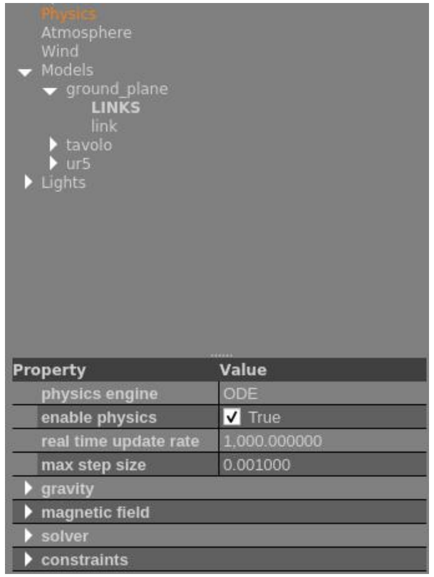
```
def talker(p):
    ros.init_node('custom_joint_pub_node', anonymous=True)
    p.pub_des_jstate=ros.Publisher("/ur5/joint_group_pos_controller/
                command", Float64MultiArray, queue = 1)
    loop_rate = ros.Rate(1000.)  # 1000hz
    q_des0 = np.array([-0.3, -0.78, -2.56, -1.63, -1.57, -1.0])
    while not ros.is_shutdown():
        p.q_des = q_des0
        p.send_des_jstate(p.q_des)
        loop_rate.sleep()
```

- register in rospy the name of your node, without it the ROS Master cannot start communicating with the node, check with **rosnode list**
- anonymous = True ensures that your node has a unique name by adding random numbers to the end of NAME.
- declares that your node is publishing to the /ur5/joint_group_pos_controller/command topic using the **Float64MultiArray** message
- The queue_size argument limits the amount of queued messages if any subscriber is not receiving them fast enough.
- creates a Rate object, it offers a convenient way for looping at the desired rate (1000 Hz). You need to be sure processing time does not exceed 1/rate time interval

Create a custom joint state publisher (rospy)

```
def talker(p):
    ros.init_node('custom_joint_pub_node')
    p.pub_des_jstate = ros.Publisher("...")
    loop_rate = ros.Rate(1000.)  # 1000hz
    q_des0 = np.array("...")

while not ros.is_shutdown():
        p.send_des_jstate()
        loop_rate.sleep()

if __name__ == '__main__':
    myPub = JointStatePublisher()
    try:
        talker(myPub)
    except ros.ROSInterruptException:
        plot()
```

- checks if your program should exit (e.g. if there is a Ctrl-C or otherwise). Equivalent of ros::ok() in roscpp
- thanks to this function, is possible to control the **simulation** speed (e.g. to slow down or speed-up) by reducing/increasing the **real_time_update_rate** in Gazebo. All nodes will be synchronized



- Used to stop the node, catches a ROSInterruptException exception, which can be thrown Ctrl-C is pressed allowing to perform some final actions (typically generating plots)

Filling the JointState message

```
def send_des_jstate(self, q_des):
    msg = Float64MultiArray()
    msg.data = q_des
    self.pub_des_jstate.publish(msg)
```

- to fill in the message, the general rule of thumb is that you can pass no arguments to the constructor and initialize the fields directly

- the publish() function is how you send messages. The parameter is the message object. The type of this object must match with the type given in the Publisher definition

- now run the ur5generic.py script to start the simulation and call the script with:

```
>python3 -i custom_joint_publisher.py
```

- you can exit the Python script and stop the node pressing CTRL+Z

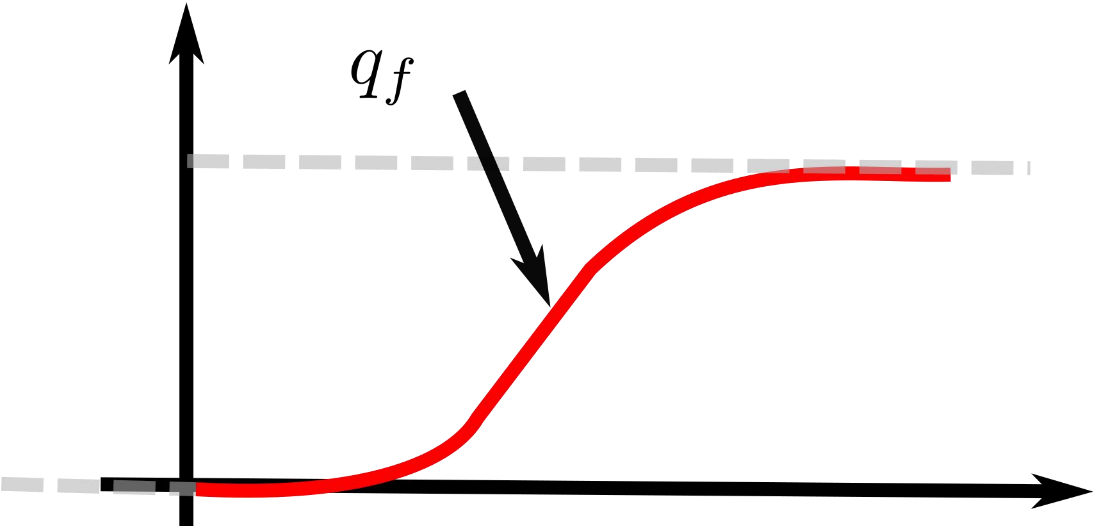
Step-reference signal

- Now we want to change the set-point at t=4.0s to reach a new configuration where we add 0.4 rad to the second joint (shoulder lift)

```
while not ros.is_shutdown():
# generate step reference
      if time < 4.:
         p.q_des =  q_des0
      else:
         p.q_des = q_des0 + np.array([0., 0.4, 0., 0., 0., 0])
```

- Despite this code works in simulation, if you run this on the **real robot** it will go in protection mode, because an **abrupt** set-point change involves velocity beyond the limits
- To avoid abrupt changes in the set-point we can **filter** the set-point designing a 2nd order filter
- A more appropriate way is to plan a smooth trajectory (your exercise)

Reference filtering




```
def initFilter(self, q):
   self.filter_1 = np.copy(q)
   self.filter_2 = np.copy(q)

def secondOrderFilter(self, input, gain):
   self.filter_1 = (1 - gain) * self.filter_1 + gain * input
   self.filter_2 = (1 - gain) * self.filter_2 + gain * self.filter_1
   return self.filter_2
```

- The gain should be tuned to have a certain settling time for the filter
- We need to **initialize** the filter with the **actual** joint configuration at the startup
- To obtainin it we need to code a **subscriber node** that subscribes to the **/ur5/joint_states** topic that is published by Gazebo (in simulation) or by the ur5 robot driver (on the real robot)

Add a Joint state subscriber

```
def talker(p):
    ros.init_node('custom_joint_pub_node', anonymous=True)
    p.pub_des_jstate = ros.Publisher("/ur5/joint_group_pos_controller/command", Float64MultiArray)
    p.sub_jstate = ros.Subscriber("/ur5/joint_states", JointState, callback = p.receive_jstate)

    loop_rate = ros.Rate(1000.)  # 1000hz
    q_des0 = np.array([-0.3, -0.78, -2.56, -1.63, -1.57, -1.0])
    p.initFilter(q_des0)
    while not ros.is_shutdown():
        if time < 4.:
            p.q_des =  q_des0
        else:
            p.q_des = p.secondOrderFilter(q_des0 + np.array([0., 0.6, 0., 0., 0., 0]), 0.002)
        p.send_des_jstate(p.q_des , p.qd, p.tau)
        loop_rate.sleep()
```

- Declaration of the subscriber that subscribes to the topic **/ur5/joint_states** with callback **receive_jstate**

```
def receive_jstate(self, msg):
    for msg_idx in range(len(msg.name)):
        for joint_idx in range(len(self.joint_names)):
            if self.joint_names[joint_idx] == msg.name[msg_idx]:
                self.q[joint_idx] = msg.position[msg_idx]
```

Note: In rospy, each subscriber has its own thread which handles its callback functions automatically, so there is no need of spinOnce() in Python (only in C++)

```
self.joint_names =  ['shoulder_pan_joint', 'shoulder_lift_joint', 'elbow_joint', 'wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']
```

Assignment

- Now it is time that you implement your own controller manager...
- Translate the python publisher/subscriber in C++ to set the reference for the sine trajectory
- Test the code is working running the simulation with the ur5_generic.py script
- Hint 1: you can run the ur5 generic commenting the line p.controller_manager.sendReference(p.q_des, p.qd_des, p.g) to avoid having two publishers publishing on the same topics and set the control_type = 'position'.
- Hint 2: to manage vectors and matrix in robotics, a very popular library is **Eigen**

Eigen tutorial: https://dritchie.github.io/csci2240/assignments/eigen_tutorial.pdf

Locosim installation

- Docker container (Linux, Mac): follow the **tutorial** at: github.com/mfocchi/lab-docker
- Virtual machine: www.dropbox.com/sh/5trh0s5y1xzdjds/AACchznJb7606MbQKb6-fUiUa
- Software repository (Linux, MAc): github.com/mfocchi/locosim (native installation)

Useful resources on ROS:

- Books: www.dropbox.com/sh/5uloeo2qqgjf18x/AAAJxTlnlwC2mm-Xh7Ce3UU3a
- Forums: answers.ros.org/questions/
- Wiki: wiki.ros.org/it
- Tutorials: www.theconstructsim.com